RL-TR-97-231
Final Technical Report
February 1998

AUTOMATING OBJECT-ORIENTED SOFTWARE DEVELOPMENT FOR PARALLEL PROCESSING SYSTEMS

Arizona State University

Stephen S. Yau, Changju Gao, Debin Jia, Jun Wang, Jiazheng Wu, and Bing Xia

**AIR FORCE RESEARCH LABORATORY
ROME RESEARCH SITE
ROME, NEW YORK**

APPROVED: [illegible]
JOSEPH CAVANO
Project Engineer

FOR THE DIRECTOR: [illegible]
WARREN H. DEBANY JR., Technical Advisor
Command, Control & Communications Directorate

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE February 1998	3. REPORT TYPE AND DATES COVERED Final Jul 95 - Dec 96

4. TITLE AND SUBTITLE AUTOMATING OBJECT-ORIENTED SOFTWARE DEVELOPMENT FOR PARALLEL PROCESSING SYSTEMS	5. FUNDING NUMBERS C - F30602-95-C-0202 PE - 62702F PR - 5581 TA - 20 WU - PL
6. AUTHOR(S) Stephen S. Yau, Changju Gao, Debin Jia, Jun Wang, Jiazheng Wu, and Bing Xia	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Arizona State University Computer Science and Engineering P.O. Box 875406 Tempe AZ 85287-5406	8. PERFORMING ORGANIZATION REPORT NUMBER N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/IFTB 525 Brooks Road Rome NY 13441-4505	10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-97-231

11. SUPPLEMENTARY NOTES

Air Force Research Laboratory Project Engineer: Joseph P. Cavano/IFTB/(315) 330-4033

12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A set of computer-aided software engineering (CASE) tools for the software development framework for parallel processing systems based on the PaRallel Object-Oriented Functional computation model (PROOF) toward the automation of object-oriented software development for parallel processing systems is presented. An object-oriented analysis tool has been developed to identify and express parallelism in the problem statement. We have developed a communication estimation tool to estimate the communication among objects, and a clustering tool to partition the objects in groups so that the intergroup communication is reduced and concurrency with specified user requirements is realized. The parallelism analysis tool further explores potential parallelism by analyzing the invocation relations and data flow among objects. The PROOF/L back-end translator is extended to support clustering and dynamic allocation features in order to achieve better performance on a workstation cluster. The software effort using our approach can be greatly reduced due to implicit synchronization and communication, the user-friendly graphical interfaces of the CASE tools, the automated object-oriented analysis and parallelism analysis. The performance of software developed using our approach can be improved due to the integration of the object clustering algorithm, the parallelism analysis tool and the extended back-end translator.

14. SUBJECT TERMS Parallel Processing Systems, Object-Oriented Software Development Framework, CASE Tools, PROOF, PROOF/L, Object-Oriented Analysis, Clustering Algorithm, Back-End Translator, Parallelism Analysis, Large-Scale Parallel Software Development			15. NUMBER OF PAGES 122
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Although parallel processing is becoming popular in scientific computation, decision-support and financial analysis activities, sequential computers still remain dominant in practice. Although it is cost effective to use sequential computers for those applications which are inherently of sequential nature, there are many applications whose structures are inherently parallel and which require a large amount of computing power to complete the necessary computation in a short time interval, such as command, control, communication and intelligence (C3I), telecommunication, simulation, space science, and weather forecasting. To effectively utilize various parallel processing systems, we need to have a software development methodology in which the design specification is architecture independent and the architecture dependent issues, such as partitioning (clustering), grain size determination and allocation, can be either automatically taken care of in the compilation (or translation) or addressed at a later stage of software development so that the design specification is more re-usable and the developed software is more portable.

One promising approach to identify and express parallelism in software development for parallel processing systems is the object-oriented paradigm which naturally reveals existing parallelism in the problem space. In addition to modifiability, maintainability and reusability, this paradigm is better than others in that the concept of an object can be used at earlier stages of software development cycle than the implementation stage. It implies that parallel processing aspects, such as parallelism and communication among parallel components, can be naturally handled at an earlier stage of software development. Consequently, it is easy for the programmer to handle parallelism and communication among parallel components. The functional paradigm is based on the concept of functions. This paradigm offers great potential for exploiting parallelism by removing side effects caused by assignment statements. The simplicity and effectiveness of the functional programming style and the mathematical basis of functional programming languages are demonstrated by functional programming languages, such as FP [1], SML [2] and SISAL [3]. We have developed the computation model PROOF (PaRallel Object-Oriented Functional) [4] in which the object-oriented paradigm is integrated with the functional paradigm to take advantage of many useful features of both paradigms. We also developed a software development framework for parallel processing systems based on the computation model PROOF [5, 6, 7, 8], and developed a front-end translator to translate PROOF/L code to IF1, and back-end translators to translate IF1 code to nCube C and KSR C [8].

In order to exploit parallelism at the lower level, we have to consider the trade-off between the communication overhead and parallel execution. The communication overhead due to excessive parallelism may degrade the performance significantly. Thus, we need to find proper size of partitions or grains so that the completion time of the program execution can be minimized. The existing back-end translators do not incorporate any object partitioning algorithm. In order to make our software development approach for parallel processing system to yield software with better performance, we need to incorporate the result from an object clustering algorithm in the back-end translation so that parallelism among objects can be effectively exploited.

In this project, we have conducted the following tasks:

1. Investigation on object identification methods.

   We have investigated Booch's [9], Shlaer & Mellor's [10], Coad and Yourdon's [11], and Rumbaugh's [12] approaches in their applicability to the object-oriented software development for parallel processing systems, identify the advantages and disadvantages of these methods, and develop a set of guidelines on how to use these approaches for software development for parallel processing systems in conjunction with our software development approach. However, the methods used in those approaches to identify objects are similar. Our method for identifying objects is the combination of Booch's and Rumbaugh's approaches [13].

2. Development of CASE tools.

   - We have identified the requirements for developing an integrated CASE tool set for supporting the object-oriented analysis phase of our software development approach.
   - We have developed a graphical user interface to help the user specify the relation among objects graphically. This graphical user interface has the features of object creation, object annotation, object specification, object deletion, object copying, and others. It also supports the hierarchical representation of objects so that it can be used for large-scale object-oriented software development for parallel processing systems. This graphical user interface is based on X-window running on Sun workstations [14]. All the basic functions of the graphical user interface are implemented and tested. The source code for this interface is written in C.
   - We have developed a communication estimation tool and object clustering tool with graphical user interface. The tool creates the graphical representation for the object hierarchy, allows user to specify communication costs, estimates the unspecified communication costs and clusters the objects into groups using our object clustering algorithm. This tool is based on X-window running on Sun workstations. All the basic functions of this tool are implemented and tested. The source code for this tool is written in C.

- We have developed a CASE tool to help the user identify and express parallelism in the problem statements, and evaluate parallelism to determine the benefits of using parallel processing systems. This CASE tool shares the information with the graphical user interface, and helps the developer identify the types of objects, and analyze the parallelism among objects by producing a list of each type of objects. All the basic functions of this tool are implemented and tested. The source code for this tool is written in C. This tool doesn't require X-window system.

3. Incorporation of the result of our object-partitioning algorithm to the back-end translator. We have implemented the object-partitioning algorithm and incorporated the results to the back-end translator in order to exploit parallelism among objects. By doing so, we are able to utilize the parallel processing systems more effectively and reduce the development effort.

4. Demonstration of the guidelines for identifying the objects in the problem statements, the CASE tools and our software development approach.

   We have demonstrated the guidelines for identifying the objects in the problem statements for software development for parallel processing systems with some examples. We have demonstrated the CASE tools on a cluster of Sun workstations by developing some applications. We have also demonstrated the speedup of our approach, especially the effectiveness of the object-partitioning algorithm and the back-end translator by developing and running some application software on a workstation cluster.

Chapter 2

Our Approach to Software Development for Parallel Processing Systems

2.1 Overview of Our Approach

Our approach to software development for parallel processing systems is based on the computation model PROOF which incorporates the functional paradigm into the object-oriented paradigm [7]. Our framework [6], as shown in Figure 2.1, consists of the following phases: object-oriented analysis, object design, partitioning, PROOF/L coding, front-end translation from PROOF/L to IF1, grain size analysis, back-end translation from IF1 to a target language of a parallel processing system, and allocation.

In this project, we have developed CASE tools to support object-oriented analysis, object-oriented design, and object clustering. We have developed a back-end translator which translates IF1 to C on a workstation cluster. We have also designed and implemented algorithms for analysis of parallelism among objects and allocation of objects. The object clustering tool uses the output from the object-oriented analysis tool and provides graphical output which will help the users develop parallel applications. The tools have been tested using various examples. We will present the tools with examples in the next several chapters.

In object-oriented analysis, the requirements are decomposed into a set of interacting objects. We have developed a semantic analysis tool to support object-oriented analysis. The concurrent/parallel aspects of the system behavior are first visualized using the object-communication diagrams, and then analyzed and specified using the graphical user interface. The objects identified in the object-oriented analysis phase are then designed and verified in the object design phase. In the partitioning phase, the objects in the software system are partitioned into a set of clusters to improve the overall performance of the software system by minimizing communication cost and exploiting parallelism among objects. The front-end translation, grain size determination and back-end translation are grouped together by a dotted rectangle in Figure 2.1, called *transformation*, where the architecture-independent PROOF/L code is transformed into a target code to be allocated into the parallel machine. The architecture dependent issues need not to be considered until after the front-end translation. In the grain-size analysis phase, the proper sizes of tasks are determined using the architecture-dependent information, such as communication cost and execution time in the target parallel machine. In this transformation, the partitioning and grain size analysis results are incorporated to generate the target code which can be efficiently executed on the target parallel machine. After the target code is generated, it is allocated to a set of processors. We have developed CASE tools to facilitate object-oriented analysis and design for parallel processing systems. In the following sections, each phase in our approach is summarized [8]. The object-oriented analysis and design phases are partially automated using our CASE tools.

2.2 Object-Oriented Analysis

Our object-oriented analysis is different from other object-oriented analyses, such as Rumbaugh's OMT(Object Modeling Technique) [12] and Coad & Yourdon's Object-Oriented Analysis [11], in that our approach focuses on the concurrent/parallel aspects of the system, but other approaches do not address concurrency explicitly. Our approach starts from the given requirement statements. The requirement statements often contain ambiguities. When ambiguities are found during the object-oriented analysis, we report them to the user or domain expert to clarify the requirement statements. Thus, our object-oriented analysis is an iterative process which continues until all the functionalities are satisfactorily specified. When the requirement statements are not complete, we may use the guidelines given in [15] to clarify them although more research in this area is needed.

The object-oriented analysis phase consists of the following steps [5, 6, 7]:

1) Identify *objects* and *classes*.

2) Determine *class interfaces*.

3) Specify *dependency* and *communication relationships* among objects.

4) Identify *active*, *passive* and *pseudo-active* objects.

5) Identify the *shared* objects.

6) Specify the behavior of each of the objects.

7) Identify bottleneck objects, if any.

8) Check the completeness and consistency.

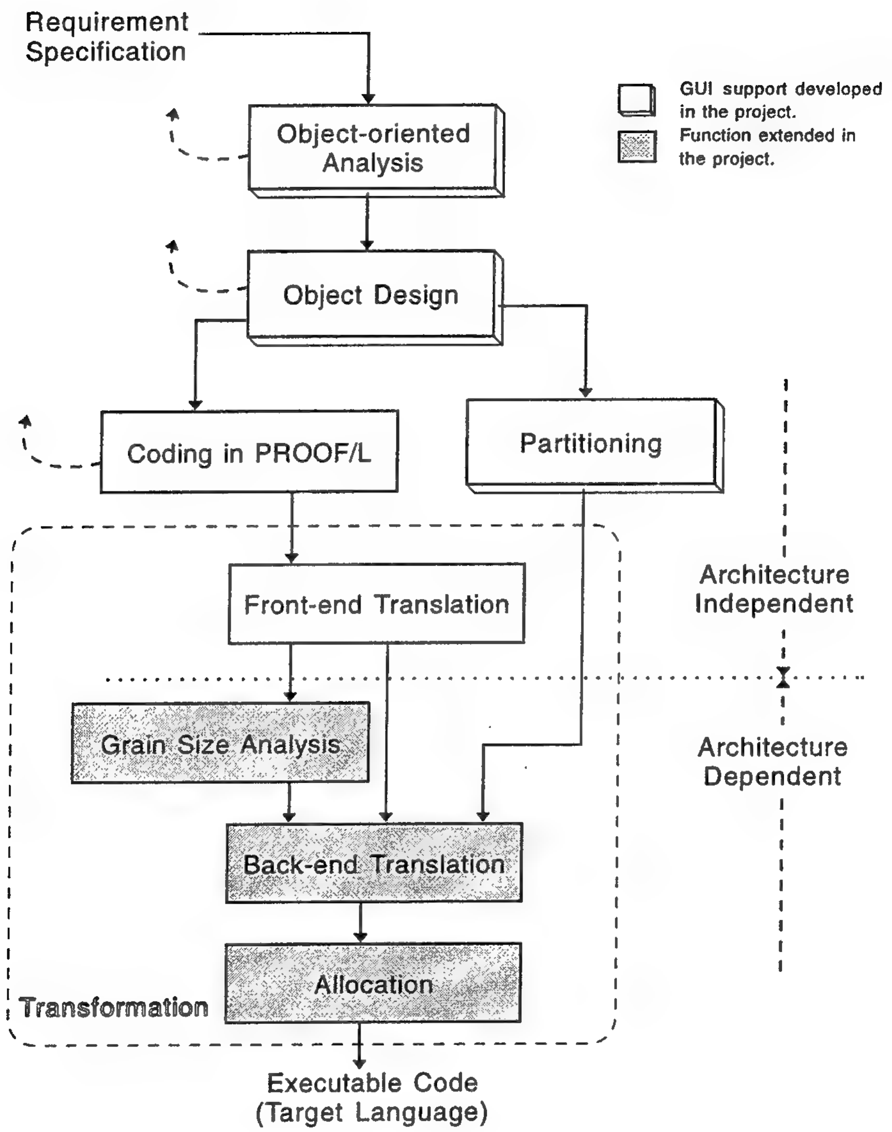


Figure 2.1: Our PROOF software development framework for parallel processing systems.

In Step 1), the software system is represented by a set of communicating objects. Objects are identified by analyzing the semantic contents of the requirement specifications. All physical and logical entities are recognized. Each object corresponds to a real-world entity, such as sensors, control devices, data and actions.

In Step 2), object class interfaces are determined. In PROOF, every object is considered as an instance of an object class. Instead of defining objects directly, the object classes to which they belong must be defined. Class interfaces may consist of both local methods and global methods. The local methods are class specific methods; while global methods can be accessible to any other global methods, any method of other classes, or body of any object instance. The purpose of global methods is to provide a flexible way to address general operations which do not belong to any specific classes.

In Step 3), the static relationship among objects are specified using the object communication diagrams, in which the objects are represented as rectangles, the links between the objects (which can be specified as method invocations) indicate the communication between objects, and the arrows on the links indicate the directions of communications.

In Step 4), the objects are classified according to their invocation properties as *active, passive* or *pseudo-active.* An active object can initiate activation of other objects by invoking methods of other objects. The methods defined in an active object cannot be invoked by other objects, but they can be invoked by other methods defined within the active object itself. A passive object is activated only when its methods are invoked by other objects. Pseudo-active objects can invoke the methods of other passive or pseudo-active objects and also has methods which can be invoked by other active or pseudo-active objects. All the threads of control in the application start from the active objects. We can identify all the possible threads of control and then use this information to check for the completeness and the consistency of the analysis.

In Step 5), once the static structure of the software system is determined, we identify shared objects from them. A shared object has local data which can be accessed by a number of objects. The shared objects can be further divided into *read-only* shared objects and *writable* shared objects. The read-only object has local data which cannot be modified by other objects. The writable object has local data which can be modified by other objects. Read-only objects can be freely duplicated as many times as desired. All the access to the data in the writable shared objects needs to be synchronized to maintain the consistent status of the data.

In Step 6), the behavior of each object is specified using the following notations:
- SEQ($m_1, m_2, \ldots, m_n$): The methods $m_1, m_2, \ldots, m_n$ are executed sequentially.
- CON($m_1, m_2, \ldots, m_n$): The methods $m_1, m_2, \ldots, m_n$ are executed concurrently.
- WAIT(m, O): Object is waiting for the invocation of its method m by another object O to proceed with its execution.
- SEL($m_1, m_2, \ldots, m_n$): The object selects one of the methods for execution from among the methods $m_1, m_2, \ldots, m_n$.
- ONE $-$ OF($WAIT(m_1, O_j), \ldots, WAIT(m_n, O_k)$): The object permits only one of its methods $m_1, \ldots, m_n$, to be invoked by other objects. ONE-OF construct is used

in cases where other objects could try to invoke the methods defined in the object O simultaneously, while the object O permits only one object to invoke its method at a time.

In Step 7), bottleneck objects which may unnecessarily degrade the performance of the software system are identified. Usually, a bottleneck object be a shared writable object. One can identify a shared writable object from the description of the object behavior in Step 6). If such an object is found, then redo or refine the analysis to reduce the bottleneck if possible.

In Step 8), the result of the analysis is verified with the user requirements. From the given user requirements, the possible threads of controls are identified, and each of them is examined using the behavior of the objects specified in Step 5).

For more detailed information on the object-oriented analysis in our approach, refer to [7].

2.3 Object Design

Objects obtained from the analysis phase have to be designed. In our approach, the object design is specified using the notation defined in PROOF/L [4]. The class interface definitions and information about the object behavior are used to design the objects. Our approach to object design involves the following four steps:

1) Establish the class hierarchy.

2) Design the class composition and the methods in each object.

3) Design the bodies of the active and pseudo-active objects.

4) Verify the object design.

In Step 1), since some common operations and/or attributes between the objects may not be apparent in the analysis phase, different objects are reexamined to identify the commonality between the classes in the design phase. A set of operations and/or attributes that are common to more than one class can then be abstracted and implemented in a common class called the *superclass.* The subclasses then have only the specialized features.

In Step 2), the composition and the methods for each object class are designed. The class definition consists of *composition* and *methods.* The composition defines the internal data structure of the class. Various constructors, such as list and Cartesian product, are provided. A typical functional style is adopted in the method definition. A rich set of functional forms, i.e. high-order functions, as well as primitive functions are predefined. In the method design, the internal state of the object to which the method belongs is included as both the input and output parameters so that side-effects are avoided. A method of an object consists of an optional *guard* and an *expression.* The guard is a predicate specifying synchronization constraints and the expression statement specifies the behavior of the method. The object which invokes the method is suspended when the value of the attached guard is False, and it is resumed when the guard becomes True. The guard attached to a method is defined in a way that it only depends on the status of the local data, and does not depend on the definition of any other methods. The global methods which are class-wide methods should also be specified here for determining the properties of the operations.

In Step 3), a body is associated with each active and pseudo-active object. There is no body associated with a passive object as it does not invoke any methods. The role of a body is to invoke a method and to modify the state of the objects represented by their local data. The body in each object is expressed in the form $e_1//e_2//\ldots//e_k$ where each e_i is an expression representing method invocations and expressions separated by // are evaluated simultaneously. // is a parallel construct indicating parallel execution. The modification of an object is expressed by the *reception* construct which has the form $R[|o|]e$, where o, called a *recipient object*, is an object name and e is an expression with applications of purely applicative functions only. The reception construct can occur only in the bodies of active and pseudo-active objects. The reception construct indicates that the object o receive the value returned as a result of evaluating the expression e. This construct modifies the states of the object.

In Step 4), the design of the objects done in the previous phase has to be verified and analyzed. For this purpose, we transform our design into Petri nets [16], which have been selected in our approach mainly because our design can be easily represented in a Petri net model and because many techniques have been developed to analyze Petri-net models. The transformation of our design to Petri nets consists of the following three steps: 1) transformation of bodies to Petri nets, 2) composition of the nets, and 3) refinement of the nets.

For more detailed information on the object-oriented design steps in our approach, refer to [7].

2.4 Partitioning

In the partitioning step, the objects in the software systems are partitioned into a set of clusters in order to reduce communication cost among processors while maintaining the parallelism among the objects. It is very difficult to achieve linear speedup due to communication costs among processors, contention of shared resources and inability to keep all the processors busy [17]. That is one of the reasons that there is a large gap between the ideal peak performance and the real performance in most parallel computers. The partitioning approaches for reducing communication cost are divided into three categories: graph-theoretic [18, 19], integer programming [20, 21] and heuristics [22]-[23]. One of the common assumptions in these approaches is that the execution time for each module and the communication time among modules are given as input. Our partitioning approach does not assume that exact execution time and communication time are available. In addition, most of the existing partitioning approaches cannot be used when the software is decomposed as a set of such objects having shared data.

The objective of our partitioning approach is to improve the overall performance of the software by reducing communication cost among processors and increasing parallelism among objects. The input to our approach are (1) the structure of the objects in the software system, expressed using the user interface description language (Syntax and semantic definitions are in Appendix A), (2) communication information specified by the user or estimated using the communication estimation tool, and (3) the number of replications for each object as required for fault tolerance. Using this information, we represent the software system as an directed weighted graph in which every node represents a cluster of objects and every edge between two nodes has a weight representing the degree of contribution for improving the overall performance of the software system by parallel execution of the two clusters. The details of our clustering approach with illustrative examples has been presented in chapter 4.

2.5 Transformation

The transformation of the PROOF/L code to a target code involves the following steps: partitioning, front-end translation, grain-size determination and back-end translation.

The PROOF/L code is first translated into an IF1 code and then the IF1 code is translated to the target code. The former is called *front-end* translation which is a semantics-oriented translation, and machine or architecture dependent issues are not involved. The latter is called *back-end* translation.

In the grain size analysis step, we focus on finding proper grain sizes within each object. Thus, we can consider each object as an independent program. We represent the program as a directed graph in which each node corresponds to an IF1 construct, and each edge represents a data dependency relation. In order to perform grain size analysis, the execution time of the IF1 constructs is estimated statically, and the communication time between them are estimated by examining the type information of the data transmitted. The estimation can be done statically by analyzing the assembly code for these constructs. We developed efficient heuristic algorithms of three different types of parallelism – tree parallelism, graph parallelism and pipe-lined parallelism. The details of these algorithms can be found in [7].

The back-end translation is performance-oriented and machine or architecture dependent. After partitioning and the result is incorporated to the intermediate form, which is translated into corresponding equivalent target code In the previous project, we have developed two back-end translators for two target parallel processing systems, KSR and nCube. However, the translators have not included partitioning and grain size determination. Hence, we have developed a new back-end translator for C for a cluster of workstations. The translator has incorporated other new features like load balancing, dynamic allocation.

2.6 Allocation

After the target code is generated, the target code is allocated to the parallel processors in such a way that the execution time of the target code can be minimized by exploiting parallelism in the target code.

One of the problems that must be solved in order to achieve high performance of software for parallel computers is the allocation of tasks among the processors. Some of the factors that prevent the ideal linear speed-up in parallel processing are 1) insufficient concurrency and 2) high communication overhead [25]. The task allocation problem has been studied extensively [25]-[28]. In these approaches, efficient heuristic task allocation algorithms were introduced. Factors to be considered in the allocation phase include the number of processors, the number of processes to be allocated, interprocessor communication pattern, and communication overhead.

Chapter 3

Object-oriented Analysis Tool

3.1 Introduction

In this chapter, we will present a framework for an integrate tool set to assist users in object-oriented analysis for seqential as well as distributed parallel processing systems [14]. This framework emphasizes on how to easily represent OOA results textually, how to automate graphical layouts from textual representations, and how to keep consistency between textual and graphical representations. We have also integrated the verification of the OOA results [29] into the framework. This framework is demonstrated by a CASE tool set which is being developed on SUN workstations under the X window environment.

3.2 Our Approach

Our approach to overcoming the deficiency of the current CASE tools for the OOA [14] can be illustrated in Figure 3.1. The analysis tool served as the front end to any drawing tool, can assist the users to simplify the OOA specification. Then, the "automated layout support," served as a back-end support for drawing tools, can layout the corresponding graphical notations based on the OOA specification. Changes can be made interactively between users and analysis tools or drawing tools for the following purposes:

- add, delete, and update any object-oriented component.
- resolve any unreasonable placement of graphical notations.

Consistency needs to be implicitly maintained between analysis tools and drawing tools, and verification can be realized by comparing the original problem statements with the OOA specification.

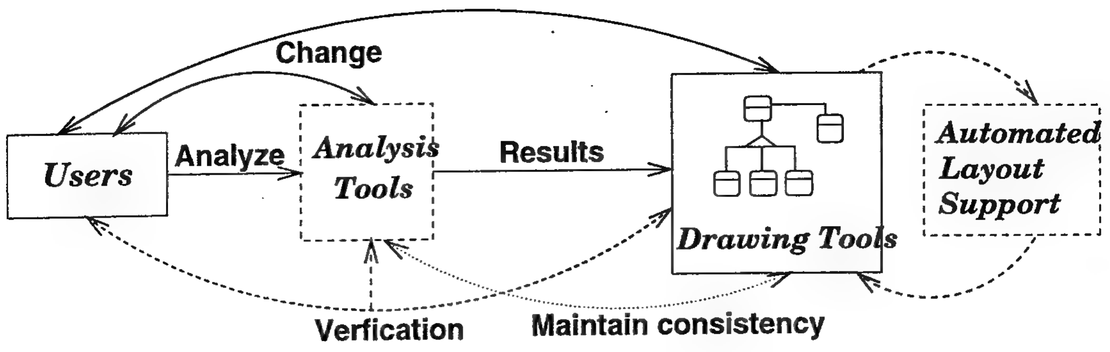


Figure 3.1: Our approach to development flows in OOA for application software development.

Similar approaches for supporting automation in structured analysis and structured design (SA/SD) have been developed [30, 31] with emphasis on automated generations of data flow diagrams and entity relationship diagrams. Since the OOA provides better modularity, abstraction, hierarchy, encapsulation than the SA/SD, our approach needs to provide more automation supports to the OOA, which can assist the users to analyze the problem statements more meaningfully.

Our approach to providing automation supports in the OOA starts from the problem statements. It consists of the following phases: semantic analysis, layout support, analysis verification, and common drawing technique. The approach is shown in Figure 3.2.

3.2.1 Semantic analysis

Our approach starts from the problem statements given in a natural language. Problem statements are assumed to be complete in the functionalities of the problem domains. The purpose of the semantic analysis is to assist users to concentrate on problem statements to derive the corresponding topological specifications in object-oriented components, including classes/objects, attributes, methods, aggregation hierarchy, inheritance hierarchy, association, events, and states.

The current CASE tools allow the users to interactively draw the specifications in terms of graphical notations to represent analysis results diagrammatically, which can be time-consuming and difficult to layout, especially when the scale of application software increases dramatically. Furthermore, modifications in the geometry of graphical notations, either add, delete, or move, can create considerable ripple effects in clarity for overall appearances of graphical notations, and cause great drawbacks for the extensive usage of drawing capabilities of CASE tools.

In order to avoid unnecessary interactive drawing, textual specification is the simplest, cleanest, and quickest way for representing the OOA results. Our approach uses the initial user problem statements as a basis to start the analysis. User's scanning through the entire problem statements allows him/her to pinpoint and highlight these appropriate object-oriented components in the textual format. Different changes

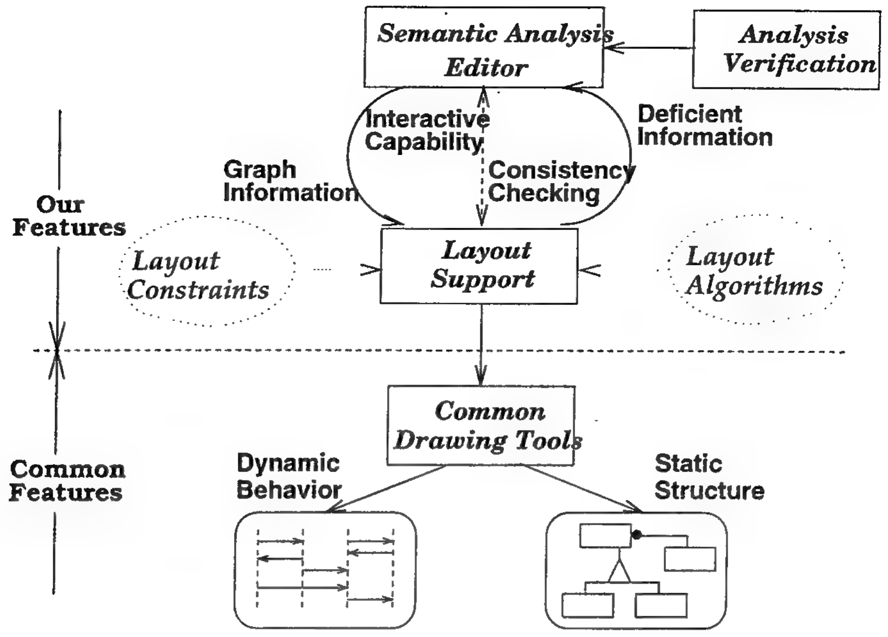


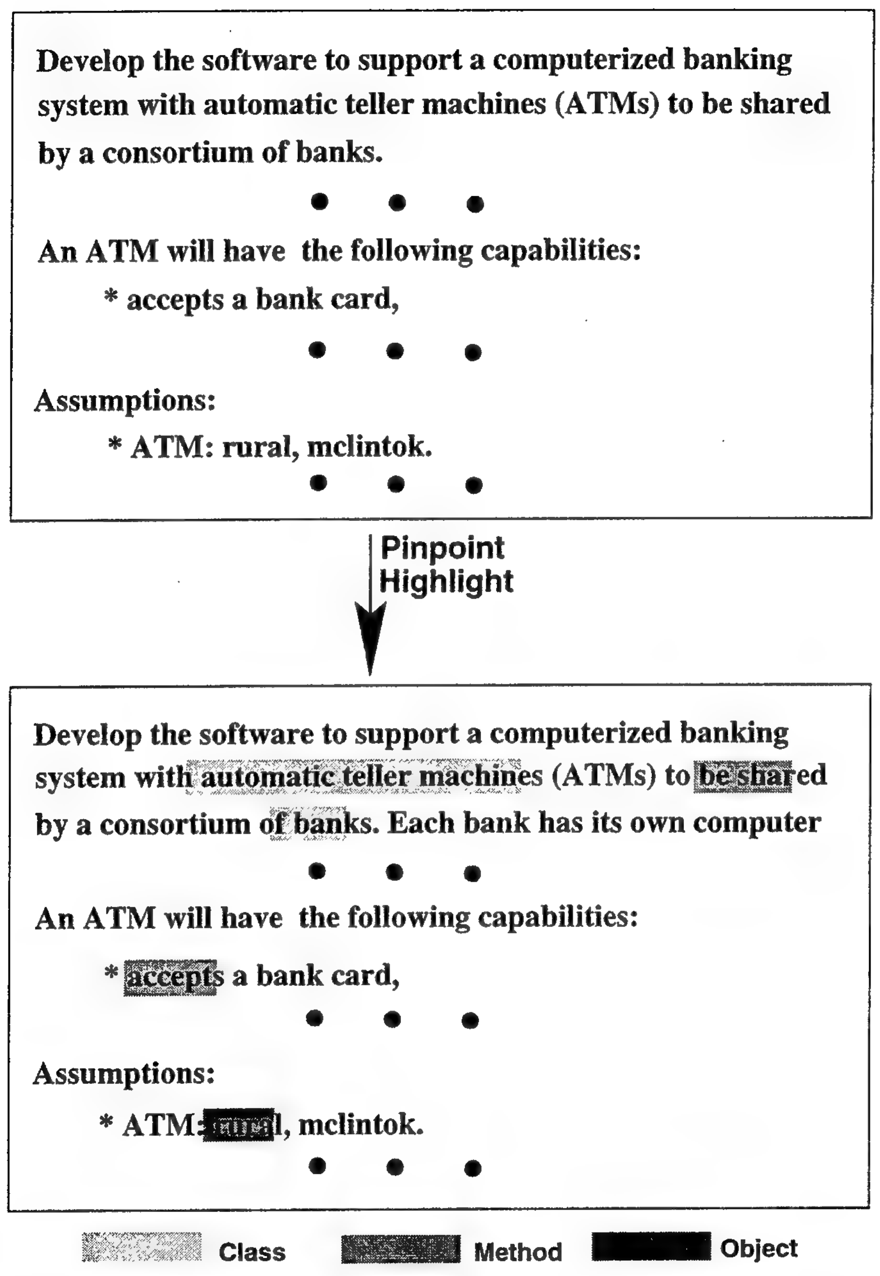
Figure 3.2: Our approach.

in highlight colors/fonts can be used for different OO components for visual aids. Any other OO components, which are not explicit in the problem statements, can be listed as the problem domain information which has some relations with other OO components. Under the semantic analysis, users can follow other popular OOA approaches, such as Rumbaugh's OMT approach [12], Coad & Yourdon's approach [11]. The process for the semantic analysis is shown in Figure 3.3. Close investigation of problem statements with user's participation is essential. The semantic analysis assists users to concentrate on problem statements in the textual format. The semantic analysis, which serves as an analysis between users and common drawing tools shown in Figure 3.1, outputs analysis results in an interface language hidden from users. Then, analysis results are passed to the following two steps: automation of analysis diagram generation and verification of analysis results.

3.2.2 Layout support

Automation in layouts determines the geometry for graphical notations organized through topological information from the semantic analysis. Either class/object diagrams or event trace diagrams can be expressed in term of a graph $G = (V, E)$, where V is the set of vertices which represents graphical notations, E is the set of edges which connects several vertices, and G denotes the entire graph information. Two standards for drawing [31] are shown in Figure 3.4.

- The straight line standard, where all connection lines are straight, and there are no explicit bends.



Develop the software to support a computerized banking system with automatic teller machines (ATMs) to be shared by a consortium of banks.

● ● ●

An ATM will have the following capabilities:

* accepts a bank card,

● ● ●

Assumptions:

* ATM: rural, mclintok.

● ● ●

Develop the software to support a computerized banking system with automatic teller machines (ATMs) to be shared by a consortium of banks. Each bank has its own computer

● ● ●

An ATM will have the following capabilities:

* accepts a bank card,

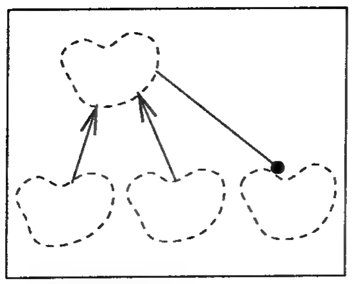
● ● ●

Assumptions:

* ATM: rural, mclintok.

● ● ●

Figure 3.3: Our process for semantic analysis.



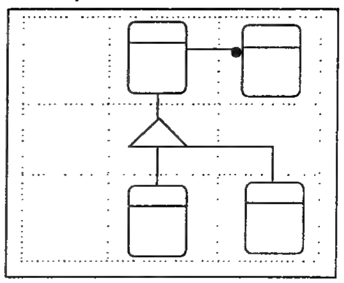


Figure 3.4: Layout standards.

- The grid standard, where all connection lines run along rectangular grids in one of two orthogonal coordinates, while nodes can be embedded in one or more grids.

Certain diagrams are viewed more naturally in one standard than in the other. As shown in Figure 3.4, Booch notation [9] appears more acceptable in straight line connections in class/object diagrams, while grid standard is more preferred in OMT object models [12].

Different from automating layouts for integrated circuit (IC) design, whose purposes are to minimize circuit areas, cross counts and total wire length, common aesthetic criterion for analysis diagrams is to keep clarity (e.g., readability) of graphical representations for the OOA, which can be determined by:

- Number of crossings between connections,
- Number of bends (wandering corners) in connection lines,
- Length of a connection line,
- Space between graphical notations,
- Distance for placement of labels on connection lines.

Priority markers need to be assigned as determined factors to resolve possible conflicts in deciding partial layouts of diagrams.

Automated layout algorithms have been used extensively to handle the early SA/SD diagrams: In [31], data flow diagrams (DFDs) and entity relationship diagrams (ERDs) can have automated layouts in grids through three phases: planarization, normalization, and compaction; In [30], DFDs can have automated layouts in grids through placement of nodes for graphical notations followed by routing paths between these nodes. In [32], directed acyclic graphs (DAGs) can have automated layouts in straight lines through four phases: assign optimal rank for nodes, order nodes from left to right, assign absolute coordinates for nodes, and route edges between nodes. The layout-automation algorithms in the existing approaches incorporate constraints based on characteristics of SA/SD diagrams or DAGs. The algorithms that we use to automate the layout for diagrams for the results from the OOA are extensions of these algorithms. Since class/object diagrams are considered extensions of DFDs and ERDs, there are more graphical notations for different types of nodes or edges in class/object diagrams (e.g., Booch or OMT notations), which create more layout constraints. We consider layout algorithms and layout constraints orthogonal to each other and they are separated as shown in Figure 3.2. The layout support bases on layout algorithms, while layout constraints for object-oriented graphical notations are declaratively enforced internally to the layout support. Since we already derived transformation rules from event trace diagrams to the state transition diagram for each class [33], we can easily obtain state transition diagrams as a part of final results.

Because the OOA is represented both textually and graphically, and textual representation is the starting point to obtain the corresponding graphical representation, consistency checking between these two representations must be maintained at all the time. Any change made in one representation invokes the proper change in the other. Besides automation in layout, interactive capabilities are also provided to users since layout algorithms have not been able to handle all the cases, and deficient information can potentially be fed back to users to decide replacement for any incomplete or unreasonable layout results.

In general, the layout support incorporates both automatic and interactive capabilities.

3.2.3 Analysis verification

We have developed a formal approach to verify the OOA [29]. Some modifications have been made to incorporate the OOA verification into our framework as shown in Figure 3.5.

The OOA can be verified in three steps:

- Transform the OOA into the formal object algebra specification.
- Build graphical representations of inheritance tree, information tree, and transition trace table based on the object algebra specification.
- Check completeness and consistency between the OOA in graphical representations and user problem statements.

Specification in the interface language from the semantic analysis is converted to the specification in object algebra.

The inheritance tree represents all the inheritance hierarchy from object algebra specification. The superclass/subclass relationship can be single or multiple. The information tree contains all the classes with attributes and methods rooted from a common

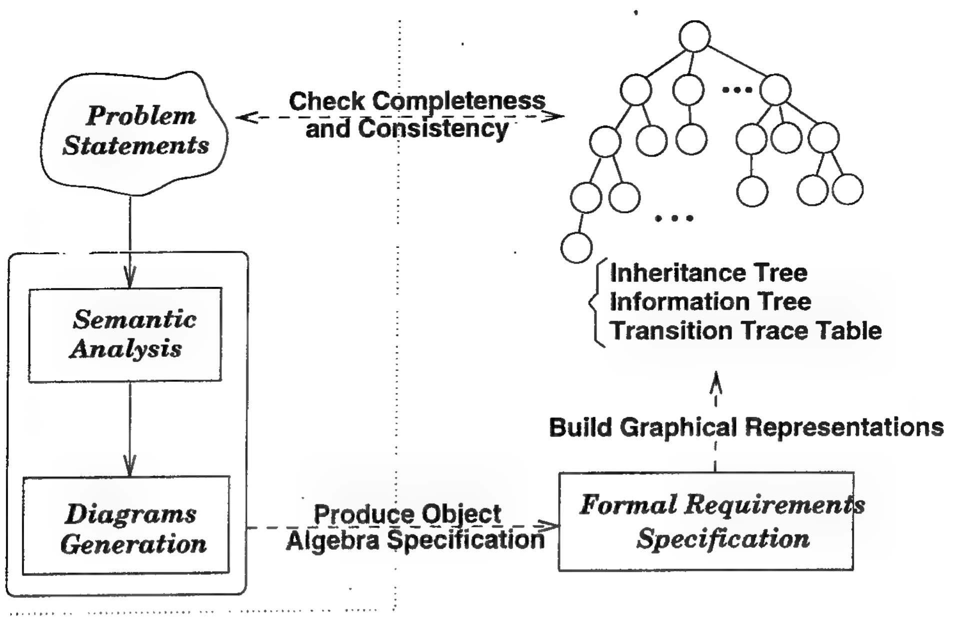

Problem Statements
Check Completeness and Consistency
Semantic Analysis
Diagrams Generation
Produce Object Algebra Specification
Formal Requirements Specification
Build Graphical Representations
Inheritance Tree
Information Tree
Transition Trace Table

Figure 3.5: OOA verification.

superclass "Software System". Associations and aggregations among classes are also specified in the information tree, as shown in Figure 3.6.

The transition trace table contains transition traces which are ordered lists of transitions between different objects. The table shows each object as a vertical line and transition as a horizontal arc from a sender object to a receiver object. Time is ordered from top to bottom. Steps to determine transition traces in object algebra are: identify relevant objects, identify initial transition, identify transitions between objects, and identify terminal transition. As a result, a transition flow diagram is generated.

Both the inheritance tree and information tree are used to verify static structures of the OOA, and the dynamic behavior can be verified through the transition trace table. In [29], systematic top-down and bottom-up approaches for completeness and consistency checks between the OOA results and user problem statements can be realized through comparisons. The difference between top-down and bottom-up is as follows:

- top-down approach dissembles problem statements and matches their components with graphical representations;
- bottom-up approach reassembles sentences from graphical representations to compare them with those in problem statements.

Both top-down and bottom-up approaches can be automated. Pattern matching can be used in the top-down approach to match partials in problem statements with graphical representations. Automatic reconstruction of sentences from graphical representations can be used to compare against problem statements.

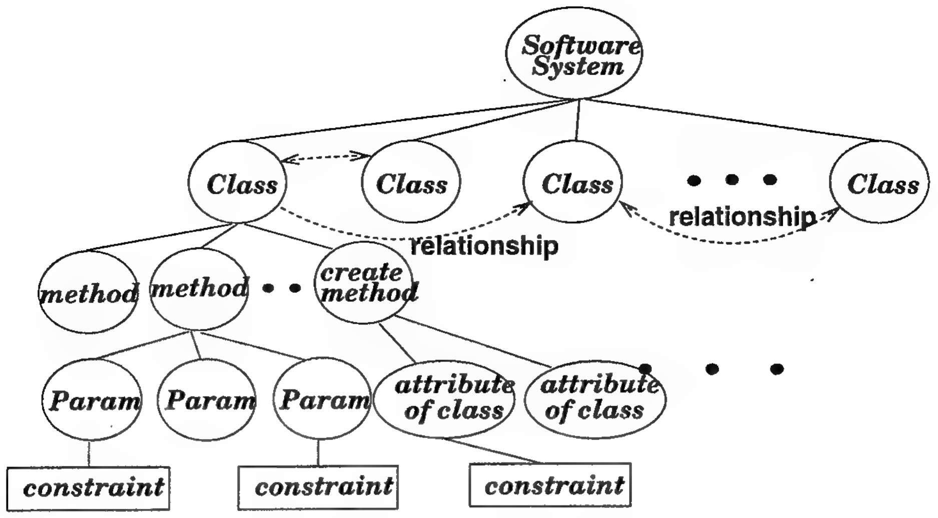


Figure 3.6: An example of the information tree.

3.2.4 Common drawing technique

Our drawing tools, like many existing OO tools, store the diagrams into the ASCII format with textual specifications of object-oriented components and their corresponding geometric locations in the display can be used following the layout support. So the outputs from the layout support can be converted to formats readable by CASE tools. Then, diagrammatic information can be obtained for users to visually verify their understanding of problem statements with those graphical representations. Afterwards, code generation capabilities in CASE tools for the later design and implementation phases can be utilized to take advantages of the benefits offered by common CASE tools.

In general, our framework can be considered as an individual framework external to other application software development frameworks or as a part of comprehensive application software development frameworks, such as PROOF model [6] and object-oriented software development framework for Autonomous Decentralized Systems (ADSs) [33]. Combining our approach with existing CASE techniques, we will be able to provide automation in analysis of problem statements and code generation in both early and later phases of object-oriented software development.

3.3 Semantic Analysis Tool

Since the emphasis for the object-oriented analysis is to effectively identify objects, the systematical way to identify objects is important for reducing the development effort for any object-oriented software. Booch [9] has claimed that any "meaningful" noun is a candidate for an object. But many nouns can also be attributes of an object. We believe that human cognitions for identifying unnecessary information for objects are critical for reducing errors.

As we describe in the report of our previous project, our analysis phase focuses on the concurrent or parallel aspect of the intended software based on problem statements, which is absent from many popular object-oriented approaches. Our object-oriented analysis is an iterative process and continues until the functionalities behind the problem statements are understood and satisfactorily specified. Since many existing popular approaches provide systematic ways to effectively capture the static model and dynamic model from the problem statements, and many of these approaches have been shown that they can assist the user to systematically derive the object-oriented analysis results, utilization of these approaches can be an effective starting point for a users who are familiar with these approaches. Reuse of any of these approaches can greatly serve the purpose of the object-oriented parallel and distributed software development.

After the derivation of the "common" object-oriented analysis results, concurrent or parallel aspect of the intended software needs to be considered and added to refine the object-oriented analysis results. Our object-oriented analysis approach not only utilizes existing results, but also provides necessary follow-on steps to assist users to insert the relevant concurrent or parallel aspect. Our object-oriented analysis phase consists of the following steps:

1. Based on the knowledge of any existing object-oriented analysis approaches, create the initial "common" object-oriented analysis results.
2. After considering the dependency and communication relationships among classes and objects, identify active, passive, and pseudo-active objects.
3. Identify the shared objects. A shared object has the local data which can be accessed by a number of objects. The shared objects can be further divided into two categories:
   - read-only shared objects, which has the local data that cannot be modified by other objects.
   - writable shared objects, which has the local data that can be modified by other objects.

   Read-only objects can be replicated many times as desired. All the access to the data in the writable shared objects can be replicated also as long as all these objects can be synchronized to maintain the consistent status of the data.
4. Check the completeness and consistency. Results of the analysis need to be verified with the user requirements. From the given requirements, the possible threads of controls are identified, and each of them is examined using the behavior of the object specified in b).

In the phase of the OOA, the purpose of the semantic analysis tool is to assist users to identify the proper object-oriented components in the problem statements. The basic frame for this CASE tool is designed with the detailed contents of the following menu selections:

```
File          Edit          Insert          Display         Outputs
  New           Undo          Class           Classes         FSL
  Open          Redo          Object          Objects           Save
  Save          Rearrange     Attribute       Inheritances      Edit
  Save as       Refresh       Method          Aggregations    Class Diagram
  Spawn                       Aggregation     Associations    Info Tree
  Spawn OODraw                Association     Events          Event Trace
  Exit                        Event           States
                                Class
                                Object
                              State
                                Class
                                Object
```

The tool is a push-button-based viewer and provides strong interactive capabilities to assist users to derive various object-oriented components with effective semantic meanings, visual, and different menu selections. The functions of the semantic analysis tools can be decomposed into four subcategories:

- Preprocess the graphical user interface specification of the problem statements to recover the previous state when the tool operates on the problem statements. Both formats in the plain ASCII and graphical user interface specification language can be recognized and properly parsed.
- Specify the display to demonstrate the problem statements in the proper visual aids, such as fonts and colors, to provide users with easy-to-capture semantic meanings of the analysis.
- Define various event-driven-type callback routines to perform functions such as adding, deleting, browsing, navigating, and updating the information of object-oriented components (classes, objects, attributes, methods, class aggregation and inheritance hierarchy, associations, events, and states), configuration changes (fonts/colors), and other management issues, such as definitions of proper constant values.
- Postprocess the tool to convert the display into the format of the graphical user interface specifications for later retrieves or refinements.

The fundamental data structure are designed for supporting manipulation of the tool. The data structure for the tool is coherently manipulated within various subcategories of the implementation and was designed using the object-oriented approach (although the code was written in C because of the interoperability across X/Motif toolkit and Lex/Yacc parsing tools) and could uniquely handle various requirements from the graphical user interface and the language processing. The *union* constructions with different identifications, which denotes different object-oriented components, can be considered as the basis for implementing the semantic analysis tool.

- Postprocess the tool to convert the display into the format of the graphical user interface specifications for later retrieves or refinements.

The fundamental data structure are designed for supporting manipulation of the tool. The data structure for the tool is coherently manipulated within various subcategories of the implementation and was designed using the object-oriented approach (although the code was written in C because of the interoperability across X/Motif toolkit and Lex/Yacc parsing tools) and could uniquely handle various requirements from the graphical user interface and the language processing. The *union* constructions with different identifications, which denotes different object-oriented components, can be considered as the basis for implementing the semantic analysis tool.

Besides the data for each entity specified in the semantic analysis tool, the global data definitions for the semantic analysis tool is required to allow the tool to be able to make references to any necessary information during the processing of the problem statements.

- Dialog boxes for generating the *entities* based on the highlighted text within the problem statements or the extra information (e.g., problem domains) provided by users.
- Dialog boxes for displaying the list of different *entities* and the information defined with each individual *entity*.
- The list of different entities, such as list of *classes*, list of *objects*, list of *associations*, list of *aggregations*, list of *inheritances*, list of *attributes*, list of *methods*, list of *events*, and list of *states*.
- The list of the *entities* provided from the problem statements and the list of the *entities* provided based on user's domain knowledge.
- The management data for determining whether the modified semantic analysis results is saved, where the highlighted text is, whether the imported information to the semantic analysis tool is the original problem statements or the last analysis results, the current working directory and file.
- To support a multi-window working environment, several semantic analysis tools can be invoked simultaneously from a single bench to process or view different problem statements or analysis results. That a semantic analysis tool to remember its own identification is necessary.

Although the code of this tool is written in the C language, object-orientation between data and their associated operations has been used during the entire implementation. For different entities, different set of functions have been specified to process different information within the entities, such as printing, associating the certain information with other entities (e.g., association of objects, attributes, methods, and states to classes). The assistance from Lex and Yacc tools under the Unix environment considerably reduces the development efforts for parsing the interface language and formal

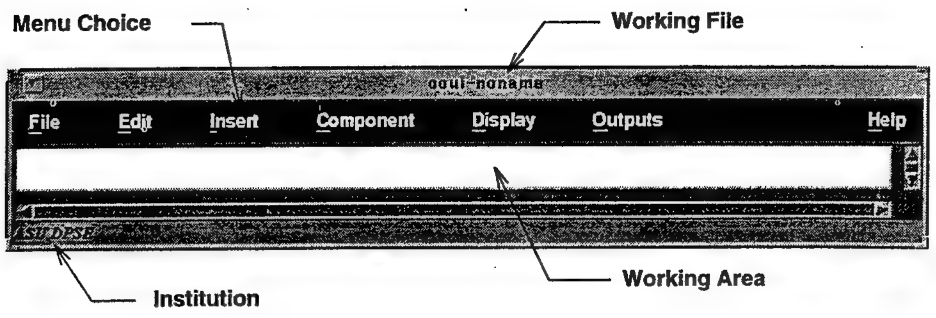


Figure 3.7: The frame for the semantic analysis tool.

object algebra specification language. The frame for the semantic analysis tool is built based on the X/Motif toolkit, which is shown in Figure 3.7.

The different menu choices has the detailed selection items described at the beginning of the chapter for adding/deleting/updating different *entities*. The browsing and navigating capabilities are also provided. The working file indicates whether there are the problem statements that are being processed. The working area is a push-button-oriented area where the problem statements can be highlighted to be specified as the different *entities* or that the information for the certain *entities* can be retrieved by clicking on the *entities*. Insertion of the domain knowledge unspecified in the problem statements can also be realized through using the proper menu choice.

The other important aspect of the implementation for the approach, the layout support tool, needs to be developed. Many algorithmic investigations have been conducted to compose some proper algorithms to be used by the tool to perform the automation in generating the diagrammatic layouts for class/object diagrams and event trace diagrams based on the textual results. Since the transformation rules have been developed for transforming event trace diagrams to a state transition diagram for each class in [33], they will be incorporated into the layout support tool.

3.4 User Interface Specification Language

In order to provide a knowledge-based repository for all the derived object-related components, we have developed a user interface specification language to record all the semantic analysis results made by users using our tools. The language is in ASCII format and similar to the hypertext which can be easily viewed by users. Proper manual modifications can be performed on the repository for recovery of any unexpected failure from the tools (e.g. accidental tool crushs). The BNF syntax of our language is presented in Appendix A.

The flex (fast lex, lexical analyzer) and yacc (syntax parser) UNIX tools have been used to parse the language. The parsing scheme of the language has been implemented. Close integration between the language and the CASE tool will be developed further.

Without proper verification of the OOA results, some errors can be hidden, which in turn will cost greatly in the later development stages, even in the maintenance stage. We have incorporated formal methods in the object-oriented concept to develop the verification scheme for the OOA for distributed systems [29], which can be used in our OOA phase.

The steps of the verification scheme are:

- Derive formal specification for object-oriented analysis.
- Convert formal specification into graphical representations.
- Systematic comparisons between graphical representations and user requirement specifications.

In order to assist developers to capture more object-related information from user problem statements, our CASE tool provides developers an underlying knowledge-base for storing and retrieving semantic meaning of user problem statements and various visual aids (e.g., font, color) from developers' inputs. The entire tool is a push-button-based view and provides strong interactive capabilities for developers to derive various object-related components with different menu selections. It is the front end to the follow-on OOA and OOD. It functions well with developers' close participations. We have completed the display aspect of the tool, such as font and color manipulations and many operations, such as file open, close, reopen, spawn (including spawn either another display of our CASE tool, or default object-oriented drawing tool, e.g. ObjectMaker, OMTool, etc.).

We have also extended our graphical user interface specification language. Besides static behavior of user problem statements, More constructs are added to represent the dynamic behavior of user problem statements. Dynamic behavior, expressed in event trace diagrams and state transition diagrams, will be described using our CASE tool prior to the OOD. Intuitions on static and dynamic behavior of user problem statements can be expressed extensively in our language and displayed on our tool. Flexibility in our language from both static and dynamic sides can significantly improve expressiveness of developers' intuitions and provide more complete information for later stage of analysis and design.

Chapter 4

Communication Estimation Tool and Object Clustering

In this chapter, we will present our approach to estimating the inter-object communication and clustering objects into groups. Our clustering approach focuses on reducing inter-cluster communication and increasing the parallelism among clusters. At the system design stage, the actual communication costs among objects are not available, we need to estimate the inter-object communication first.

4.1 Characteristics of Communication

How to effectively estimate timing for communication and concurrency is critical to our object partitioning and grain size analysis algorithms. We have investigated the effect of following three factors on inter-object communication:

- Module size, which is determined by the number of objects, complexity of execution steps in terms of sequential, IF-THEN-ELSE, and recursive/iterative control structures.
- Fan-in and Fan-out, which indicate how the input and output of any execution step are organized.
- Data and control dependencies, which explicitly express the relations among execution steps, objects and modules.

We have developed a set of tools to collect data about the relationship between the three factors and the actual communication occurring during the execution of some typical parallel applications. The major functions of these tools are:

- Tool to read the source code of a parallel application as input, and approximately determine the above three factors by analyzing the code.

- Tool to help user insert some timing code to the source program to record the communication time during the execution.
- Tool to investigate the relationship between the three factors and the communication time.

We have fed many other segments of source code into tools and get more accurate information about how module size, fan-in/fan-out, and dependencies affect communication statistically. The communication estimation will be more precise after we collect enough data in a specific application domain.

In spite of the fact that the measurement of concurrency is more difficult, we develop similar programs to investigate the relationship between the three factors of objects and the concurrency within them.

After collecting a substantial amount data about the relationship between the amount of communication and the three factors of an object, i.e. module size, fan in/fan out, and dependencies, we have the following conclusions on how the three factors affect the actual communication during program execution:

1. If an object is called by several objects, the size of data exchanged between a calling object and the called object is proportional to the size of the calling object, where the size of object is defined by the sum of the number of methods and the number of parameters.
2. If an object calls a number of objects, the size of data exchanged between a called object and the calling object is proportional to the size of the called object.
3. For a pseudo-active object, the size of data flowed in is approximately the same as the size of data flowed out.

4.1.1 Communication Estimation Algorithm

We have integrated the communication estimation tool in the CASE tool for semantics analysis of problem statements. When specifying the relationships among classes and objects, the user may easily indicate the amount of communication between classes or objects. We are also incorporating the implementation of the object-partitioning algorithm into the CASE tool, and demonstrate the analysis and partitioning results to users graphically.

The general formula for the time of transmitting a message of length n is given by [35]:

$$T_{com} = \alpha + \beta n,$$

where α is the latency, n is the size of the message, and β is the coefficent which is inversely proportional to the media bandwidth.

According to the above formula, it seems that the communication time should be a linear function of message size, but from the result we collected in 10Mbps network and 100Mbps network, we found out that the latency in the networked environment is so large that the communication time seems to be invariable if the message size is smaller than 10KByte. Subsequentially, we assume that communication cost is constant for each instance of communication in our communication estimation because of the following reasons:

- Networked workstations are becoming popular for parallel computing.
- Our experience for both 10Mbps and 100Mbps networks are true.
- Most of the messages in an object-oriented application are usually smaller than 10KB.

Hence, the total communication cost is dominated by the frequency of interactions between two objects.

We developed our communication estimation algorithm based on the above observations. We introduced a numeric value, called "trust level", for each communication frequency. The frequency assigned by users has the highest trust level because it is not estimated by the software designer.

The communication estimation algorithm can be described as follows:

Step 1 Computing the size of each object. Assign a trust level for each interaction between two objects. If the frequency has been given by the user, it has the highest trust level; otherwise, it has the lowest trust level.

Step 2 Select an interaction with the highest trust level, whose associated objects have some other interactions that have no frequency assigned. Compute the frequency according to the above observations for all the other interactions of the two objects. Decrease the trust level by one and assign it to the newly computed frequency.

Step 3 Repeat Step 2 until each interaction has a frequency assigned.

The following example illustrates our algorithm to estimate the unspecified communication costs in the directed graph shown in Figure 4.1(a).

1. Compute the object size for each of the eight objects. The user-specified frequencies of interactions from the object of size 12 to the object of size 10 and that from the object of size 8 to the object of size 6 are assumed to be 25 and 30 respectively. The corresponding trust levels are set to 1 (logically high), and the trust levels for all other interactions are set to 100 (logically low). These are shown in Figure 4.1(a).

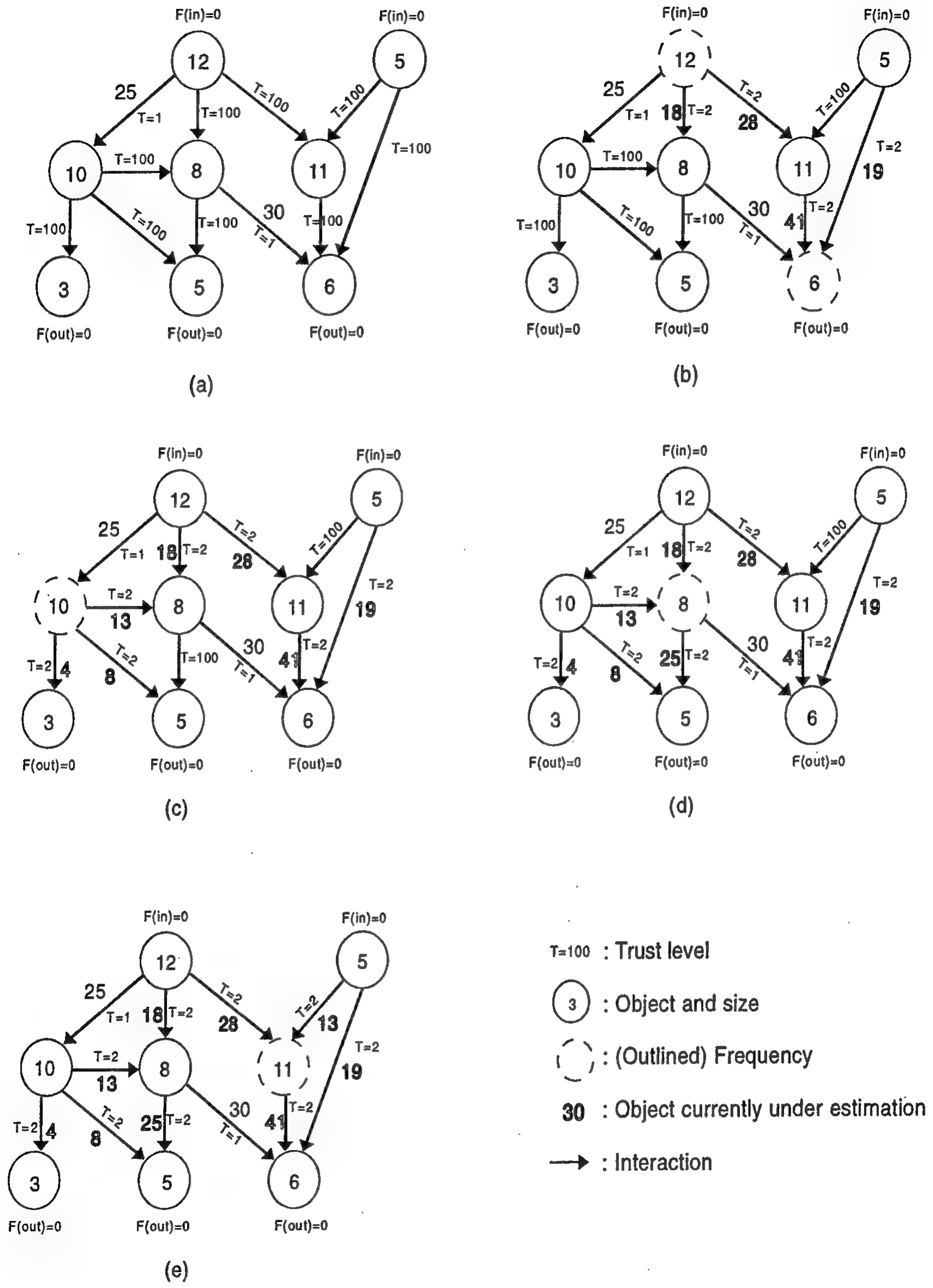


Figure 4.1: An example for illustrating our communication estimation algorithm.

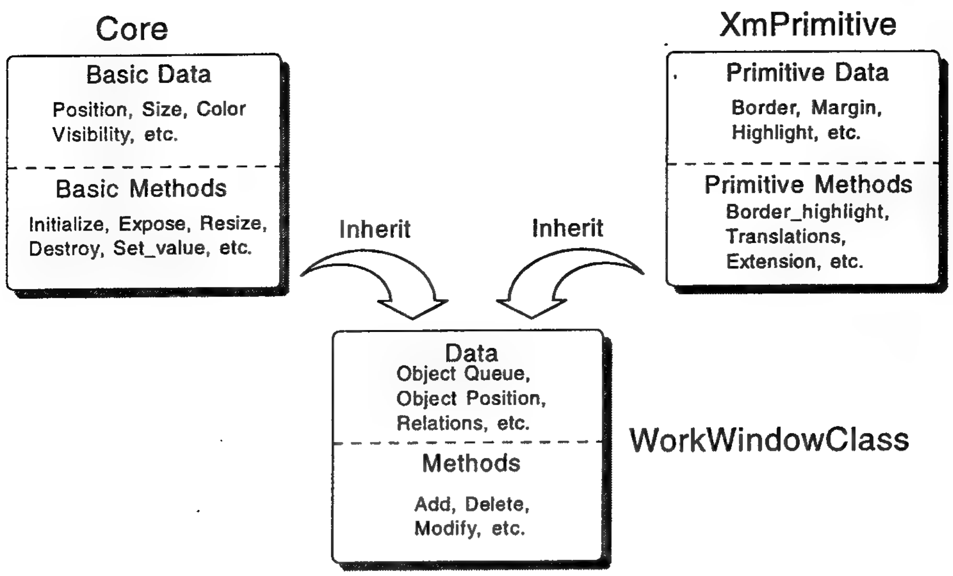


Figure 4.2: Classes and the relations among of the example.

2. Using observations 2, we find the frequencies for the interactions of the object with size 12. Using observation 1, we find the frequencies for the interactions of the object with size 6. Set the trust levels to 2 for all these interactions. All these are shown in Figure 4.1(b).

3. Using observations 2 and 3, we find the the frequencies for the interactions of the object with size 10. Set the trust levels to 2 for all these interactions. All these are shown in Figure 4.1(c).

4. Using observation 2, we find the frequencies for the interactions of the object with size 8. Set the trust levels to 2 for all these interactions. All these are shown in Figure 4.1(d).

5. Using observation 3, we find the frequencies for the interactions of the object with size 11. Set the trust levels to 2 for the interactions. The estimation process ends. All these are shown in Figure 4.1(e).

4.1.2 Communication Estimation Tool

We have implemented the communication estimation algorithm on X Window system. We defined the class hierarchy for the communication estimation tool. The X Toolkit which used to develop the Graphical User Interface supplies two basic classes - the *core* class and the *XmPrimitive* class, which are used for developing custom window classes. We used the two basic classes to construct our own custom class – *workwindowclass*. Figure 4.2 shows the classes and the relations among them.

Core is the class upon which all custom classes are based. It defines common characteristics of all classes, such as their common data, methods, and basic resources. *XmPrimitive* defines some extensions of the *core* class, such as the translation table, the flag indicating whether or not the user wants to highlight the border when the mouse pointer moves in, and the method to highlight the border. We put the local data and methods in the *workwindowclass* class. The data and methods are needed to manipulate the objects in the corresponding windows.

We also defined the X window hierarchy for the GUI tools. In Figure 4.3, each box represents the class of a window, which is followed by the name of the instance of the window. All the relations among windows are containment.

The *RootWindow* is the root for all the windows. The windows from *Shell, MainWindow, Frame*, to *WorkWindow* are used to display the objects. We used various windows to interact with the users, such as *Command Window, Message Dialog Window, Selection Box, List Box*, and *Check Box.* The *PulldownMenu* is used to pop up a menu when the user clicks a mouse button in certain areas, and the *CascadeButton* is used to show a cascade menu when the user hits the menu bar.

We have implemented the window hierarchy for the communication estimation tool, and built the major instances of the classes which were previously designed. We have developed all the *pop up* and *pull down* menus, the *dialog* boxes for input and output, and the *message* boxes for status or error notifications (e.g. warning, error reporting, etc.).

We have developed the graphical expressions for the objects, their relations, and the layout arrangement in the windows. We use a cycle to represent an object, and a numeric value to denote its size. We use a line to represent the existence of communication relation between two objects, and two numeric values associated with the line to denote the communication cost and trust level respectively as shown in Figure 4.4.

When the cursor traverses to the graphical representations of the objects and their relations in the tool, the pointer changes to a "hand", indicating that the user may press the mouse button within this area to display or modify the detailed information about the corresponding object or relation. Figure 4.5 shows the *dialog* box with the information about the object *Control Center* which is selected by the user.

Figure 4.6 shows the initial state of an example for the communication estimation tool. The question marks in Figure 4.6 mean that we do not know the communication costs for those interactions at that stage. Figure 4.7 shows what we will have after applying the estimation algorithm to the graph. Every interaction between two objects has a numeric value denoting the communication cost.

We have adopted and implemented the rules for the graphical layout of the object hierarchy. The estimation tool automatically divide the objects into layers according to their *distance* from the *root* of the object hierarchy. Here, we define the active objects as the *root* of the object hierarchy, and the *distance* is the maximum number of nodes between one object and the root. The objects in a layer be distributed equally in that layer. The tool detects if any two of the graphical expressions for the object or their relations are overlapped or too closed. If so, the tool automatically adjusts the positions of those objects at the same layer to avoid the overlapping.

Because of the limited display space, we have decided to use two ways to satisfy the

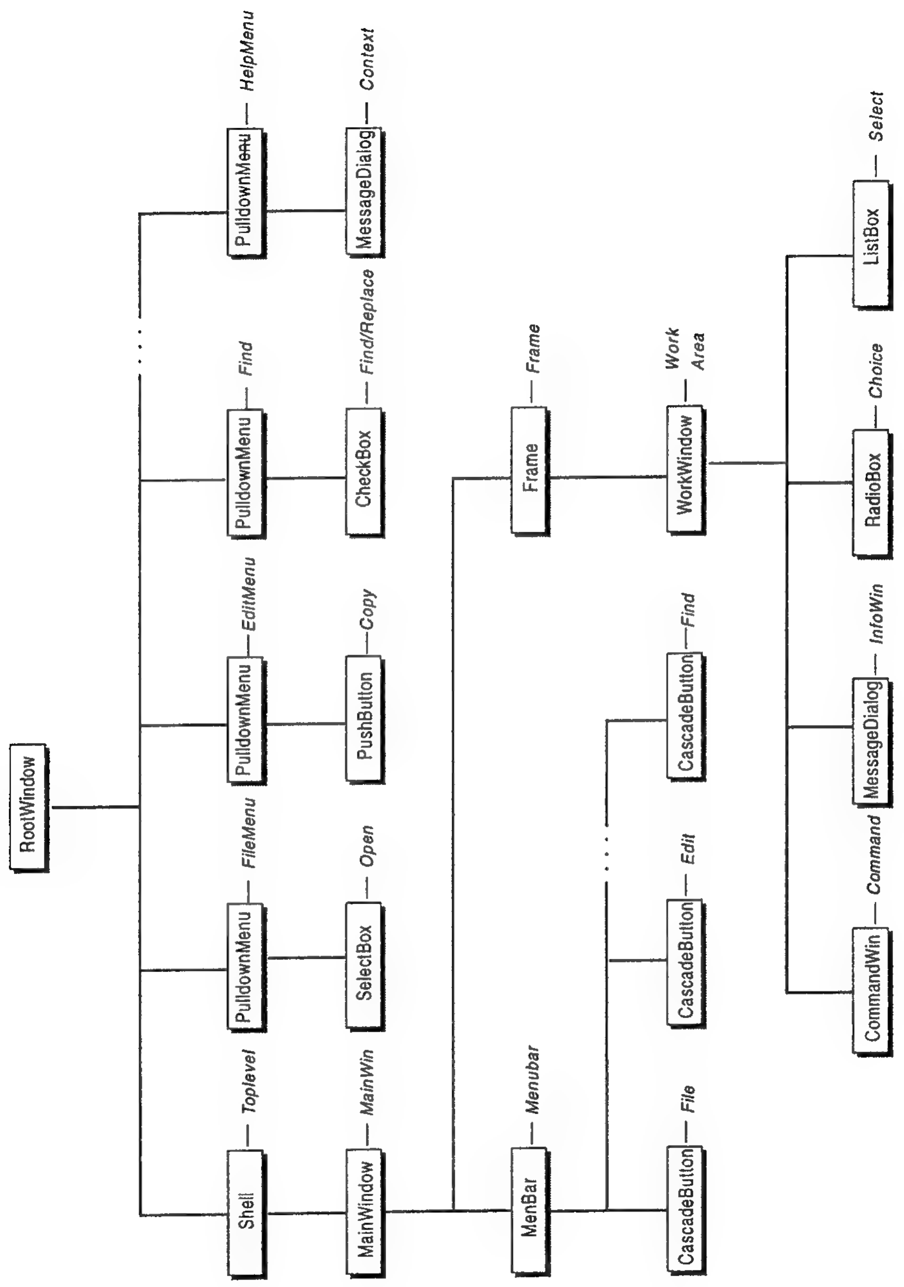


Figure 4.3: The window structure for our communication estimation tool.

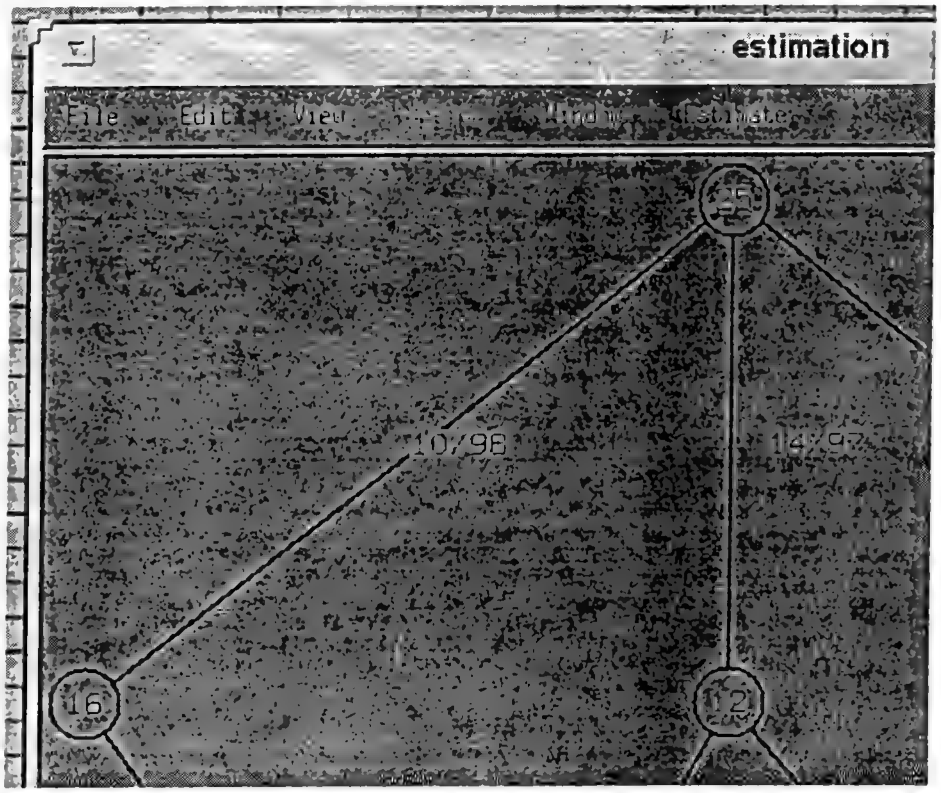
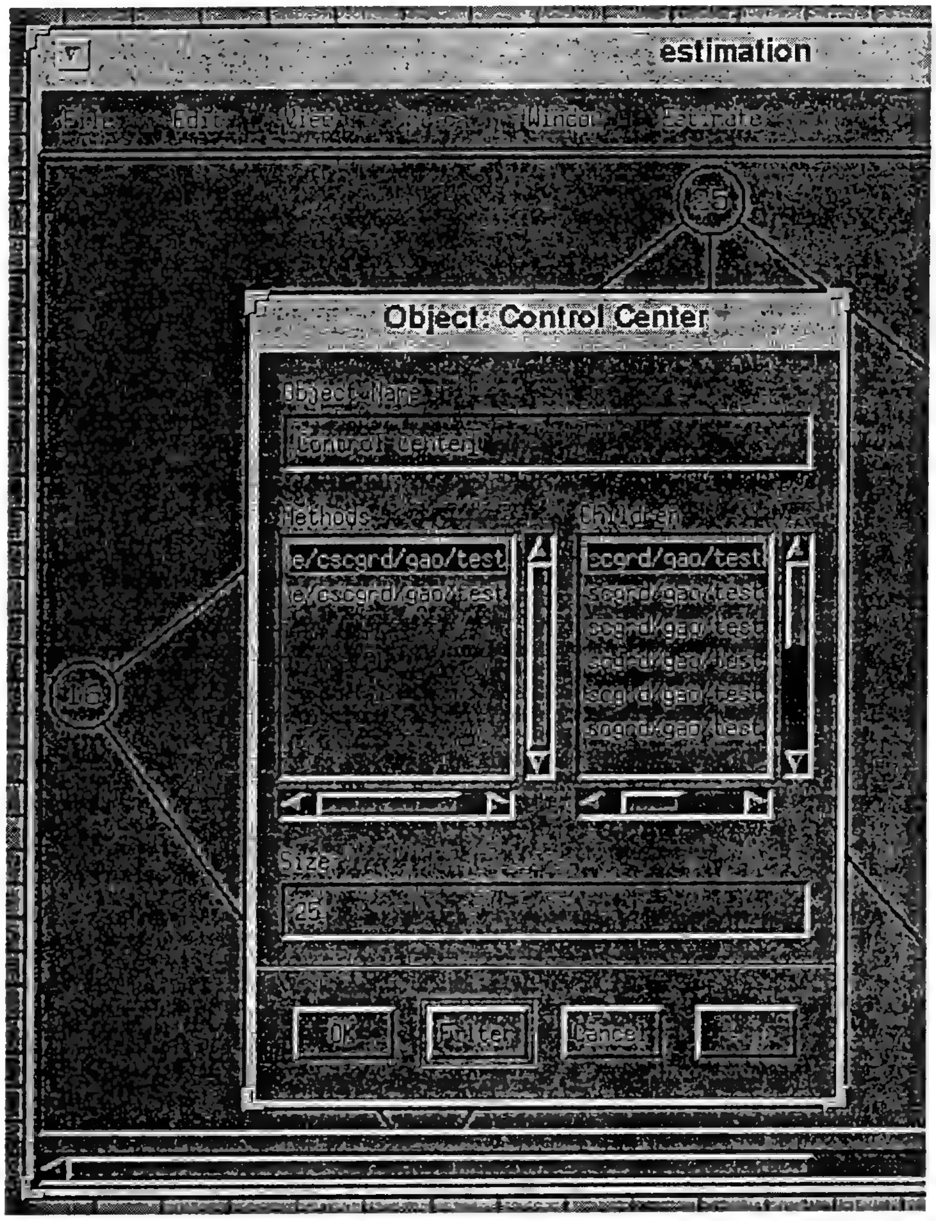


Figure 4.4: A section of the graphical expression of objects (12, 16, and 25) and their relations.



Object Name
Control Center

Methods: e/cscgrd/gao/test, e/cscgrd/gao/test

Children: scgrd/gao/test, scgrd/gao/test, scgrd/gao/test, scgrd/gao/test, scgrd/gao/test, scgrd/gao/test

Size
25

Figure 4.5: An example of the interface for modifying the object "Control Center" (25).

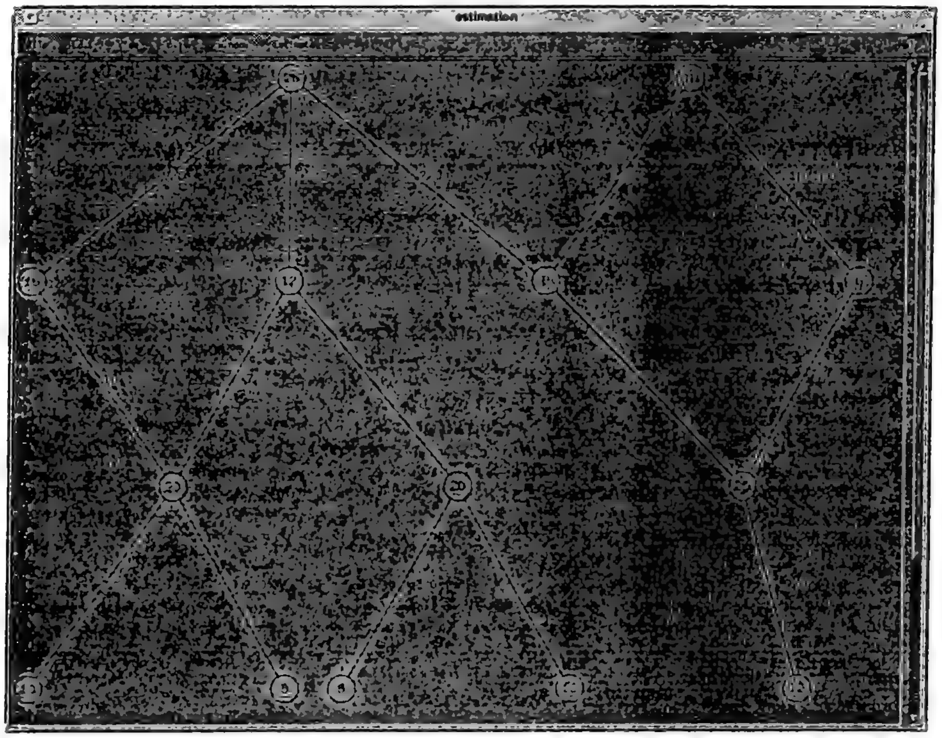


Figure 4.6: An example of the initial state (input) of our communication estimation tool.

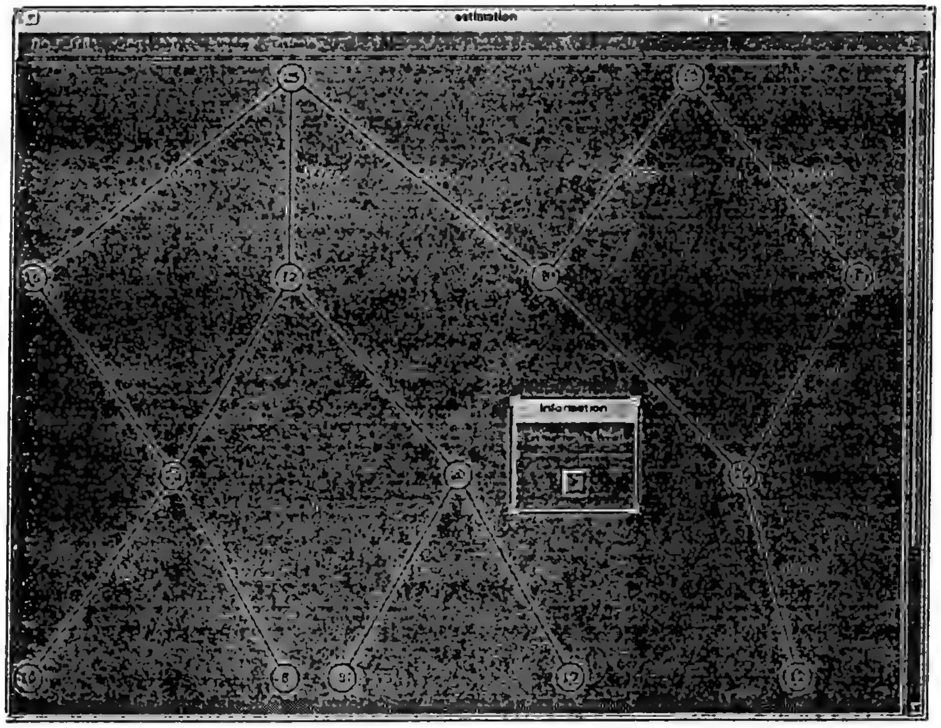


Figure 4.7: An example of the result of our communication estimation tool.

requirements for the expression of higher abstractions and detailed information. They are:

Zoom in/out The users can zoom in or out the graphical expression of objects and their relations to get the desired information.

Iconified symbols for object Objects be icons in the graphical user interface and hence we can show more objects within a fixed display space. The users can double click the icons to access the corresponding objects.

The icons which symbolize objects also help user modify the diagram easily since the users can change the information in an object through the dialog box instead of drawing tools.

Our estimation tools can show how the estimation is done by demonstrating the estimation progress step by step. The users can input or change information in the object diagram any time during the progress. They also can turn on or off the demonstration.

4.2 Object Clustering

4.2.1 Clustering Model

A parallel application is a collection of separate cooperating and communicating modules called clusters. A parallel application is modeled as a weighted directed acyclic graph (DAG) $G = (V, E, \mu, \lambda)$. In graph G, each node $v \in V$ represents an object whose execution time is $\mu(v)$, and a directed edge $e_{ij} \in E$ between node i and j represents that object O_i should complete its execution before object O_j can start. In addition, there are some kinds of data communication between object O_i and object O_j, which must happen after the completion of object O_i and before the start of object O_j. The delay incurred by the communication is $\lambda(O_i, O_j)$ if object O_i and object O_j reside on different processors; otherwise, the delay is zero. Such dependencies are due to data transfer or control transfer. In either case, the DAG establishes a precedence relation.

Figure 4.8(a) gives an example of a DAG. Each circle in Figure 4.8(a) denotes an object, and the number inside it denotes the execution time. Each arc in Figure 4.8(a) denotes a precedence relation between two objects and the number associated with it denotes the communication delay.

A cluster is a subset of node set V, and clustering is to divide V into a number of clusters. Researches have investigated two versions of clustering, depending on whether or not duplication (or recomputation is allowed). In clustering without duplication, the nodes are partitioned in to disjointed clusters and each node in the original graph is in only one cluster. In clustering with duplication, a node may have several copies in different clusters. Theoretically, for the same DAG, clustering with duplication

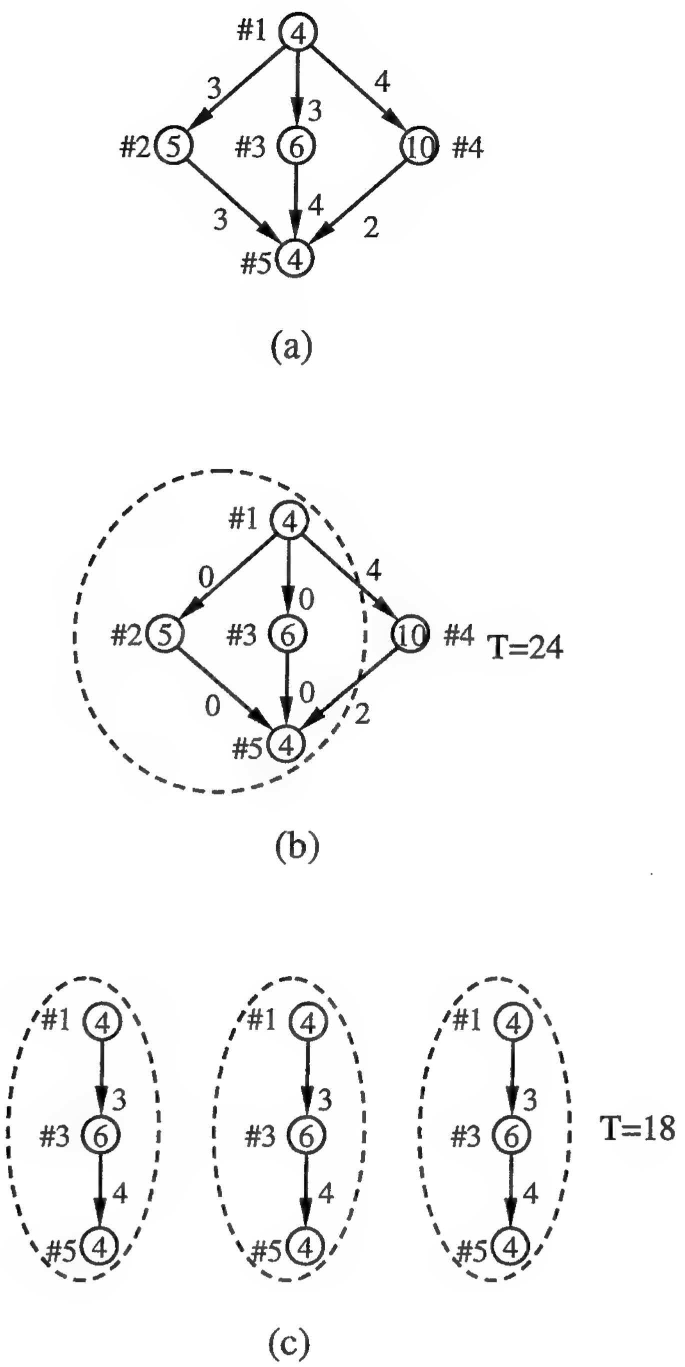


Figure 4.8: (a) The original DAG, (b) clustering without duplication, and (c) clustering with duplication.

produces a schedule with a small total execution time than that of clustering without duplication.

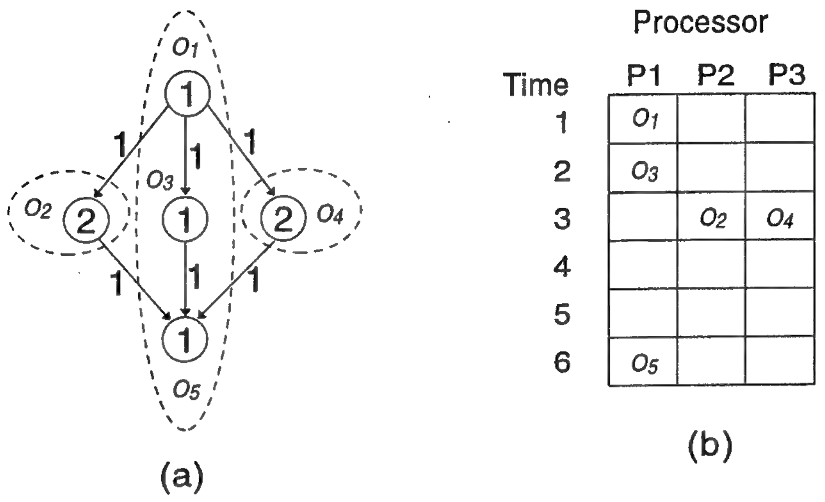


Figure 4.9: (a) Clusters of a DAG and (b) Gantt chart of the clusters in (a).

For example, for the DAG shown in Figure 4.8(a), the total execution time is 24 when duplication is not allowed, and the total execution time is 18 when duplication is allowed. In the real world, object duplication is almost impossible because these objects which need to be duplicated usually have interactions with users or I/O devices. For instance, in Figure 4.8.(a), O_1 may perform the input functions, which receive the initial data from the standard input, and O_5 may collect the results produce the visible output to the standard output. If we duplication them as in Figure 4.8(c), the user has to type the data to all the processors one by one, and he or she needs to check the results on different screens. This situation is unacceptable. So, we only discuss clustering without duplication.

A schedule of a cluster C_i is a function f

$$f : C_i \rightarrow [0, \infty)$$

which maps each object O_i in C_i to a starting time on a specific processor. The schedule should satisfy the following condition: For every immediate predecessor O_n of O_m, $O_n \in C_i$, if $O_m \in C_i$, then

$$f(O_m) \geq f(O_n)$$

otherwise,

$$f(O_m) + \lambda(O_m, O_n) \geq f(O_n)$$

A schedule of a DAG is a combination of the schedules of all the clusters. It can be represented informally using the Gantt chart. A Gantt chart consists of a list of all processors in the target system, and the lists of the start and finish time for every object. Figure 4.9(b) is an example of the Gannt chart for the scheduling of the clusters shown in Figure 4.9(a). According to Figure 4.9(b), object O_1 will be assigned to processor 1 at time 0, and object O_3 will be assigned to processor 1 at time 1.

A clustering C of DAG G is optimal if there is a schedule S of C, whose total execution time is equal or less than any other cluster C' and S' of G. The problem of finding the optimal C of S is NP-complete even when we ignore the communication delays. The large computational complexity of optimal solutions has created a need for a simplified suboptimal approach. One set of heuristics is considered better than others if its solutions are closer to the optimal solution more often. The problem addressed by our algorithm is the following: *Given a directed acyclic graph G and the number of processors which are fully connected, find a good clustering for G without clustering duplication.*

4.2.2 Our Clustering Algorithm

In this section, we will present list-scheduling heuristics for clustering a given directed acyclic graph for a parallel application. List-scheduling is the dominant method of scheduling DAGs. In list-scheduling, each node is assinged a priority, and a list of ready objects is constructed in the order of decreasing priority. Whenever a processor is available, a ready node with the highest priority is selected from the list and assigned to the processor.

The major differences among list-schedulers are their heuristics which are used to assess node priorities. The heuristics range from simple to polynomial complexity in terms of the number of nodes to schedule. The simple heuristics can schedule the graph in time $O(n)$, which the most complex ones need $O(n^4)$. Early heuristics usually ignore communication delays. Although later heuristics consider communication delays, they seldomly assign priorities according to the graph topology.

Our heuristics assign the priorities according to the execution times, communication delays, critical path, and the topology of the DAG. In our clustering algorithm, the list of the objects may not be the same for all the processors. The priorities are assigned dynamically during the scheduling for each available processors as illustrated in Figures 4.10 - 4.13.

Definition 4.1 The *minimum complete time* of a DAG is the maximum sum of execution time of the objects over all the paths from the root to the leaves of the tree.

It is obvious that the optimal execution time of the DAG is greater than or equal to the minimum complete time.

Definition 4.2 The *object level* of an object O_i in a DAG is the maximum number of objects on any path from the O_i to a terminal object.

For a given moment, the priority P of object O_i for processor j is

$$P = \alpha + T + W,$$

where α is the communication cost that be saved if object O_i is assigned to processor j, T is the execution time of object O_i, and W is the number of children of object O_i divided by object level of O_i.

α may change its value from processor to processor and from time to time, but T and W are invariable during the scheduling. Hence, we call $T + W$ *static priority* in our example.

Now, we present our algorithm as follows: *Apply the list-scheduling algorithm to the object graph using our heuristics. After the scheduling, assign all the objects scheduled on a processor into a cluster.*

```
Clustering
begin
  NOW = 0;
  All_cluster = EMPTY;
  do {
    while( processor K is available )
      assign priorities to ready objects;
      select the object with the highest priority;
      assign that object to the processor;
      put the object into cluster_K;
    end while;

    NOW++;

    if( processor K has an object assigned )
      object_complete_time--;
      if (object_complete_time == 0 )
        processor_K = available;
      end if;
    end if;

    if( object M is not currently running or finished )
      if( any of its precedors are finished )
          delay_between_the_two_objects--;
      end if;
    end if;

  } until( all objects are assigned and
```

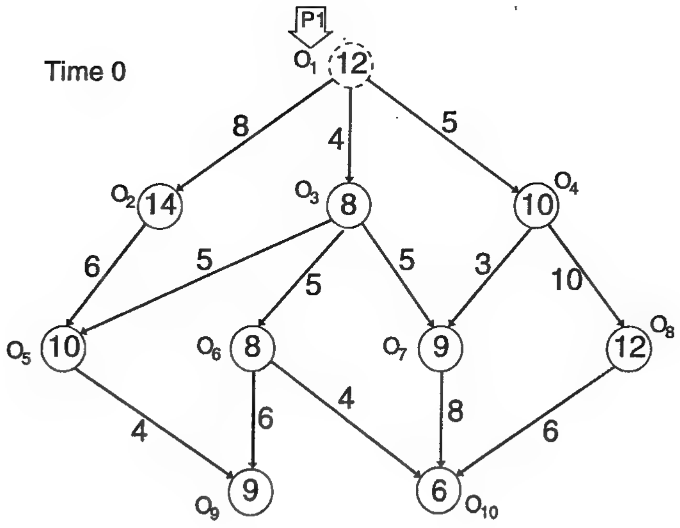


Figure 4.10: The original DAG of an example.

```
            all processor are idle );
end;
```

4.2.3 Experiments

We have implemented and experimented this algorithm in on various examples. We have developed programs to automatically produce DAGs. The number of objects of those DAGs, n, is a random number ranging from 10 to 30, and the number of edge is also a random number ranging from n to $3n$. The execution time for an object is randomly selected from 1 to 20, while the communication delay is randomly from 1 to 12. The processor number is always 4. We have applied our algorithm to more than 3,000 DAGs produced by the program. According to our experiments, the total execution time of our clustering algorithm is 1.10 times more than the minimum complete time of a DAG. Since the minimum complete time is less than or equal to the optimal clustering for any DAGs, and our results are only 10 percent more than the minimum complete times of the DAGs, it is quite obvious that our algorithm is close to the optimal.

To provide some insight of our experiments, let us consider the DAG shown in Figure 4.10. There are 10 objects and 14 edges. The static priorities for O_1 to O_{10} are: 49, 34, 29, 29, 20, 18, 16, 19, 10, and 7. At time 0, the only object ready is O_1, and hence it is assigned to processor 1. At time 12 shown in Figure 4.11, O_1 is completed. There are three objects ready for processor 1, but none of them is ready for other processors. Hence, we should construct the priority list separately for each processor. O_2 is selected for processor 1, and the communication from O_1 to O_3 and the communication from O_1 to O_4 are undergoing. At time 16 shown in Figure 4.12, the communication from O_1 to O_3 is completed, and O_3 is ready and assigned to processor 2. At time 45 shown in Figure 4.13, O_{10} is ready and assigned to processor

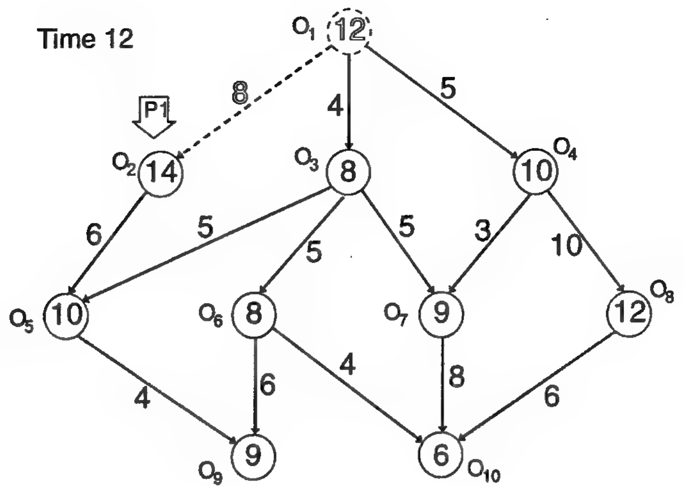


Figure 4.11: At time 12, O_1 finishes.

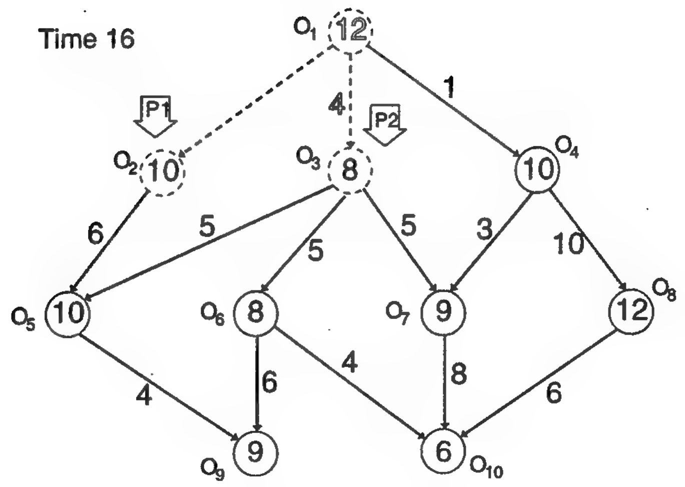


Figure 4.12: At time 16, O_2 is assigned to processor 1 and O_3 is assigned to processor 2.

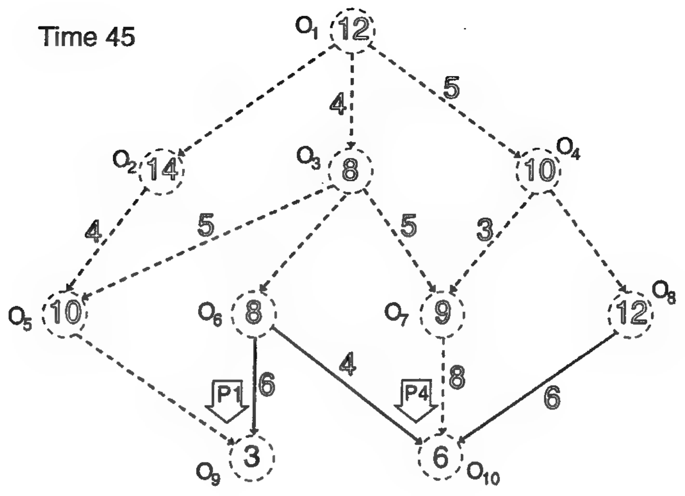


Figure 4.13: At time 45, O_{10} is assigned to processor 4.

Table 4.1: The scheduling result in Gantt chart.

Processor / Time	P1	P2	P3	P4
0	O1			
12	O2			
16		O3		
17			O4	
24		O6		
27			O8	
29	O5			
30				O7
39	O9			
45				O10
51	Done	Done	Done	Done

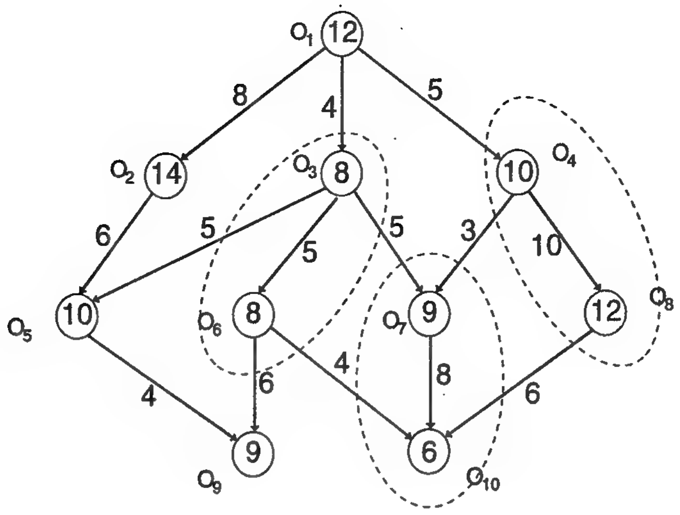


Figure 4.14: The Clustering result of the example.

4. The process of assigning objects ready to available processors is repeated until all the objects finish their executions. Finally, we get the Gantt chart for the scheduling as shown in Table 4.1. The clustering result is shown in Figure 4.14. We have four clusters for the given DAG. The total scheduling time for the optimal clustering, which for this simple example was obtained by exhaustive search over all possible schedules of all possible clusterings, is 51. The minimum complete time for this DAG is 45.

Chapter 5

Parallelism Analysis

In concurrent systems, computations are performed by cooperation of several independent execution threads. Identifying concurrency at a higher level is related to the underlying computation model and programming language used. In our approach we perform concurrency analysis in the OOA phase of the development process. The analysis result and the estimated communication cost determine clustering of objects. The difficulty to analyze concurrency before programming is that we do not have detailed information about control and data flows and data dependence. Thus, the analysis result may only discover part of the potential concurrency in the application. In this chapter, we will present an approach to analyzing concurrency between object invocations.

5.1 Our Parallelism Analysis Approach

Our approach is based on the following model:

- Object corresponds to a process. Process is the basic unit of scheduling, and thus the object is the unit for parallelism analysis. We do not consider parallelism inside the object.
- Communication between objects is in the form of Remote Procedure Call (RPC). The calling and the called processes have a Master-Slave relation, and control is passed with sending message.
- Objects are statically allocated, and no object migration occurs during the execution.

Before we present our approach, we need the the following definitions.

Definition 5.1 The *static state* of an object O is the values and links held by an object, denoted by *O.

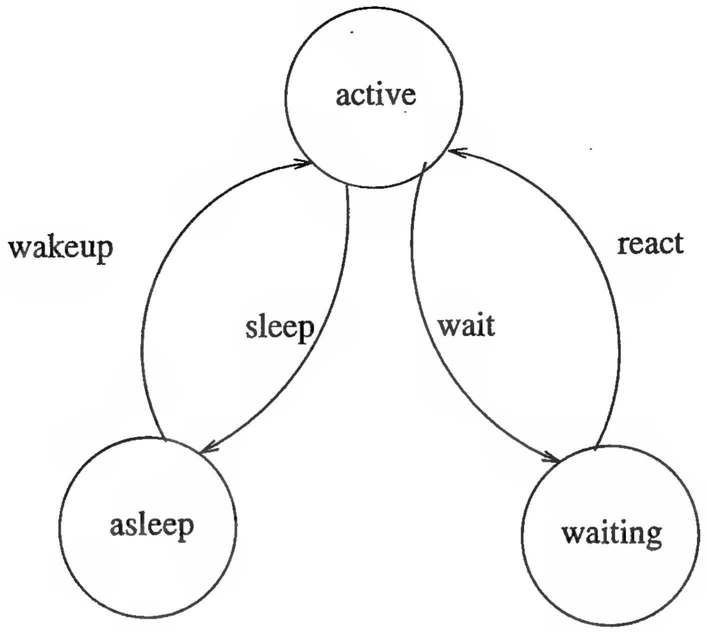


Figure 5.1: The dynamic states and critical actions of an object

Definition 5.2 The *dynamic state* of an object is the state of the object at a particular time instant. There are three dynamic states in the life of each object: An object is *asleep* when it is created, but not invoked. An object is *active* when its own method is being executed after it is invoked. After an object sends a message to other object, the object is *waiting* until the answer message (either value or control) returns.

The critical actions of an object corresponding to its dynamic states are: An object is said to *wakeup* when the object becomes *active* from *asleep* upon an invocation. It is said to *sleep* when it becomes *asleep* from *active* upon finishing executing its invoked method. An object is said to *wait* if the object becomes *waiting* from *active* when it invokes another object. An object is said to *react* if the object becomes *active* from *waiting* when it receives the return message from an object it invoked.

Figure 5.1 shows the dynamic states and critical actions of an object.

Definition 5.3 An *invocation I=([*O',M'],[*O, M])* means the method *M'* of object *O'* invokes method *M* of object *O* when *O'* is in state **O'* and *O* is in state **O*.

Definition 5.4 The *task T(I)* of an invocation *I=([*O',M'],[*O,M])* is all the operations in the control thread between the instant that *O wakesup* and the instant that *O sleeps.* A *subtask* of *T(I)* is defined as all the operations in this control thread between the instant that *O waits* and the instant that *O reacts.*

Figure 5.2 gives an example to illustrate these definitions.Initially all objects O_1, O_2, O_3, O_4, O_5, O_6 are in *asleep* state. Now, O_1 receives an invocation *I*, it wakes up and becomes *active* and remains *active* until it invokes O_2, and then it *waits* and becomes *waiting*. After a return message comes back to O_1, it *reacts* and is in *active* state again until it invokes O_6. After return message from O_6 comes back, O_1 continues

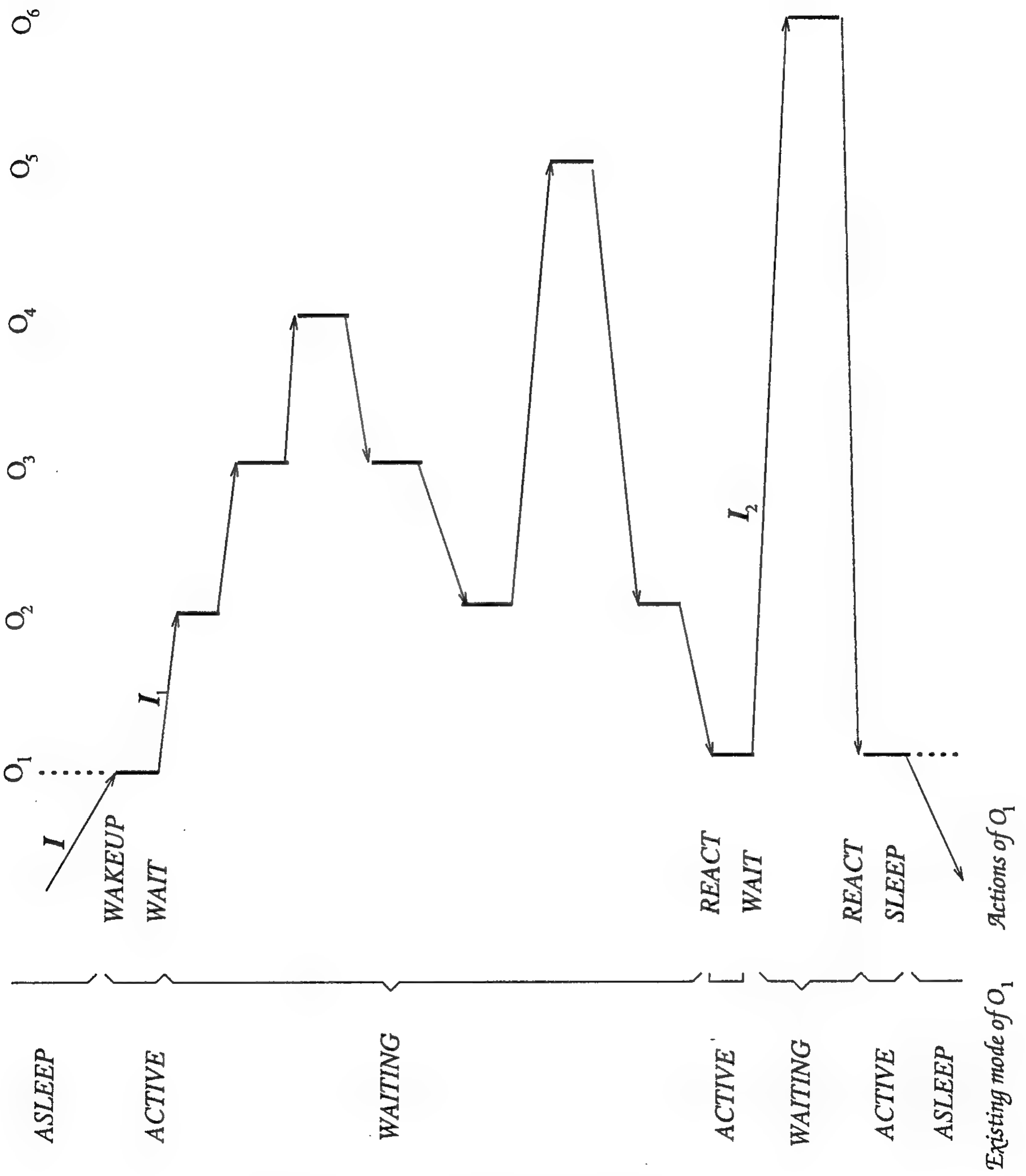


Figure 5.2: Invocation and object state of object O_1

its execution, sends back message to its invoking object and then *sleep* and becomes *asleep*. $T(I_1)$ is the task with invocation I_1, and $T(I_2)$ is the task with invocation I_2. They are two subtasks of $T(I)$.

Now we would like to present our approach to identify potential concurrency. For an invocation $I=([*O,M],[*O',M'])$, the static state of O preceding that O makes the invocation and begins to *wait* is denoted by $(*O)(I)$; the state of O following the completion of the invocation and the beginning to *react* is denoted by $(I)(*O)$. Then, for invocations from O, such as

$$I_1=([(*O)(I_1),M],[*(O_1),M_1]),$$
$$I_2=([(*O)(I_2),M],[*(O_2),M_2]),$$

if $(*O)(I_1)$ does not depend on $(I_2)(*O)$ and $(*O)(I_2)$ does not depend on $(I_1)(*O)$, the two invocations are independent of each other and can then be executed concurrently, or I_1 and I_2 can only be sequential according to the dependency relation.

To analyze the dependency of two invocations in the design phase, assume that we have invocations

$$I_1=([(*O)(I_1),M],[*(O_1),M_1]),$$
$$I_2=([(*O)(I_2),M],[*(O_2),M_2]).$$

There are three possible relations between I_1 and I_2.

- If I_1 involves O as a client and O_1 as a server, and I_2 involves O as a client or an agent, then I_2 depends on I_1. This is denoted by $SEQ(I_1, I_2)$.
- If $O_1=O_2$, we have $SEQ(I_1,I_2)$ *or* $SEQ(I_2, I_1)$.
- In all other cases, we have $CON(I_1, I_2)$.

The following is the formal description of our approach to identifying potential concurrency:

```
$T=\{ T(I) | I=(SYSTEM, ACTOR)\}$;
While $T$ is not empty{
Take a task $T(I)$ from $T$;
Find tasks in $T(I)$ which have multiple subtasks;
Let the set of such tasks be $\{T(I_j)|j=1,m\}$;
While $T(I_j)$ is not empty{
Take the first task $T(I_j)$ from it;
Let $I_j=([*O',M'],[*O,M])$;
Let Subtasks of $T(I_j)$ be $\{ T(I_k)|I_k=([*O,M],[*O_k,M_k]), k=1,n\}$
For any pair of subtasks $T(I_k)$ and $T(I_l)${
If $(*O)(I_k)$ does not depend on $(I_l)(*O)$ and
 $(*O)(I_l)$ does not depend on $(I_k)(*O)$ and
 $O_j$ is not equal to $O_l$
then $CON(I_k,I_l)$;
else $SEQ(I_k,I_l)$ or $SEQ(I_l,I_k)$
  according to the order they appear in specification;
}
}
}
```

5.2 Implementation

We identify potential concurrent objects according to the object state, event trace and method descriptions of the OOA result. The invocation precedence, and data exchanging features of the invocations are the essential elements for the identification and expression of parallelism.

The results of our object-oriented analysis (OOA) and object-oriented design (OOD) are expressed in the object algebra specification language [29]. The specification covers static model and dynamic model of the system. Static model includes description of classes, objects, inheritance, attributes, relationships and methods. The dynamic model is about those aspects of a system that are concerned with tie and changes. Dynamic modeling deals with flow of control, interactions, and sequencing of operations in a system. The major dynamic modeling concept is events, which represent external stimuli and states, which represent values of objects.

We use the following representations in concurrent object analysis:

- State transition diagram

  The dynamic model consists of multiple state transition diagrams, with one state transition diagram for each object. A state transition diagram relates events and states. When an event is received, the next state depends on both the current state and the event. A change of state caused by an event is a transition. A transition can be expressed as follows:

  $$\text{Transition } (\text{S} \times \text{E(obj)} \rightarrow \text{S'} \times \text{E}_1(\text{obj}_1) \times \text{E}_2(\text{obj}_2) \times \dots \times \text{E}_n(\text{obj}_n)),$$

  where S is the state of the object being altered by the transition, E is the input event that causes the transition and obj is the object which requests the input event. S' is the state after the transition. E_1, E_2, ...E_n are output events which are generated by the action associated with the transition, and obj_1, obj_2,..., obj_n are the destination objects.

  From the textual specification result of OOA and OOD, we can establish such a state transition diagram for each object. Figure 5.3 shows the state transition

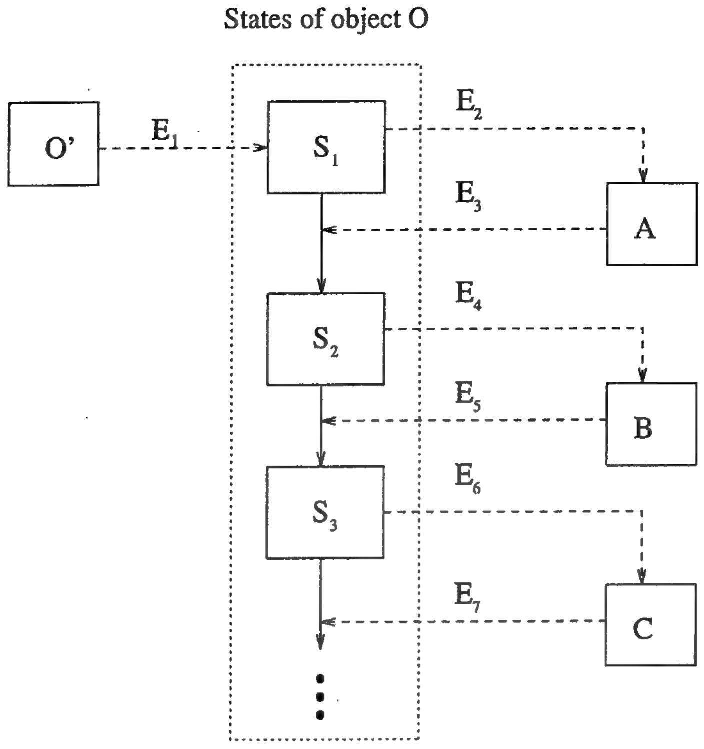


Figure 5.3: The state transition diagram of object O.

diagram of an object O.

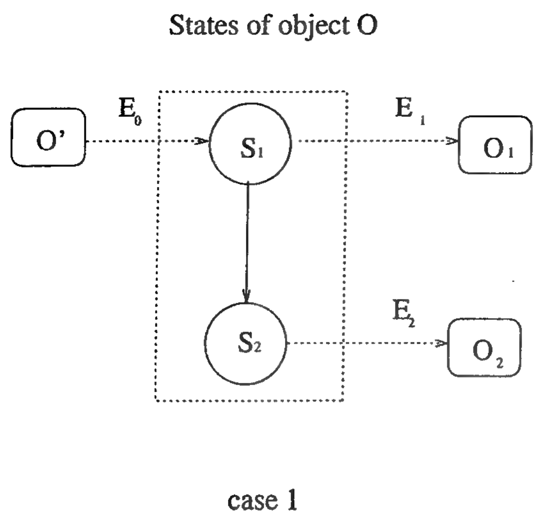
- Transition trace

  Another representation of dynamic behavior is transition trace. While the state transition diagram describes the state transitions of an object, a transition trace describes a system-wide event sequence. A transition trace is an ordered list of transitions between different objects.

  To establish a transition trace, we have to first identify initial transition. The initial transition of each class is one with no input event. A transition with input event means it shall be triggered by another object. After identifying an initial transition, we find the objects invoked by the actions in the initial transition and the transition in the invoked object state transition diagram which has the output event of the initial transition as the input event. Then, considering this transition as the initial event, continue the above process until the terminal transition is reached. Thus, we can establish the event trace. The transition trace is also an event trace, since what link the transitions are the transition events. Figure 5.7 shows a transition trace for a scenario of an elevator system in Section 5.3.

- One-way and two-way invocation

  Because of the information hiding and encapsulation of object-oriented technology, it is not desirable or possible to analyze the data dependence inside an object. Thus, the nature of invocation remains to be the major clue for parallelism analysis.



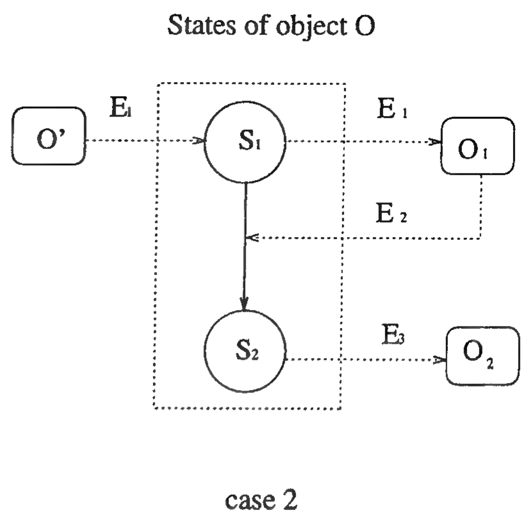


Figure 5.4: Two state transition cases for object O.

one-way invocation is an invocation which just passes parameters to another method, but does not require data back. We denote it as $\rightarrow$.

two-way invocation is an invocation both passes parameters to and requires some data back from another method. We denote it with $\leftrightarrow$

one-way trace is the consecutive *one-way invocations* in an event trace together form a *one-way trace.* We denote a *one-way trace* as $\Rightarrow$.

two-way trace is the consecutive *two-way invocations* in an event trace form a *two-way trace.* We denote it as $\Leftrightarrow$.

con of two objects O_i and O_j implies that O_i and O_j can be invoked in parallel. We denote it as $con(O_i, O_j)$.

The *one-way* or *two-way* invocation is important for dependency analysis. The $\Leftrightarrow$ part in an event trace is the most restrictive part for parallel execution. For example, we have such a *two-way trace*: $O_1.M_1 \leftrightarrow O_2.M_2 \leftrightarrow O_3.M_3 \leftrightarrow O_4.M_4$, and we want to analyze the parallel execution possibility of O in another event with objects in this event. The existence of $con(O, O_1)$ depends on the existence of $con(O, O_2)$, $con(O, O_3)$, and $con(O, O_4)$. That is whether O can be in parallel with an object O_1 in the *two way trace* depends on whether O can be in parallel with all O_i's after O_1 in the *two-way trace.*

To identify concurrency from the OOA and OOD result, we explore the state transition diagram of each object. In the state transition diagram of object O, two states S_1, S_2 are logically consecutive if there is a transition which begins at S_1 and ends at S_2. Assume that there is an event E_0 that makes O to enter state S_1 and produce an output event E_1, E_1 will invoke another object O_1 by invoking O_1's method M_1.

There are two cases of how an object's output event from a state is related to the event from the logically preceding state as shown in Figure 5.4.

Correspondingly, we identify concurrent events and objects that can be invoked concurrently in these cases.

In the first case, the output event E_2 from S_2 does not depend on the output event E_1 from S_1. These two events are from the consecutive states of one object. Therefore, we can say that objects O_1 and O_2 can be invoked concurrently by object O.

The second case is E_1 invokes O_1 in a *two-way* mode or E_3 occurs upon receiving the event E_2 from O_1, either from O_1 directly or from other object in the event trace from O_1. E_3 may depend on the return value. We need to check the event which E_3 invokes in object O_2. If the following invocation from E_3 is a *one-way* invocation, this means that it does not need to pass data to the invoked event. Only in this case, we can say that O_1 and O_2 can be invoked concurrently.

5.3 An Example

Consider an elevator system in a building, which consists of an elevator, summon buttons, summon lights, destination buttons, destination lights, arrival lights, overweight sensor, floor sensor, motor, door and a controller [36]. One scenario of actions is given as follows:

When the elevator goes to a floor, the floor-sensor is activated. Then, it sends a signal to the controller to tell the elevator's arrival. The controller turns on the arrival light, and checks the summon and destination light of the floor. If there is a passenger who wants this floor as a destination, then the destination light should be turned off. If there is a summon from this floor, the summon light should be turned off. Then, the controller stops the motor, and tells the elevator it is safe to open the door. The elevator then instructs the door to open itself. After the door is closed, the elevator checks the overweight sensor to see if it is overweight. If it is not overweight, then it informs the controller that it is now ready to receive movement instruction. Upon receiving the elevator ready signal, the controller checks the destination light and summon light status to get the next destination relative to this floor. Then the controller gives instruction to the motor to go.

The scenario is shown in Figure 5.5 including only the objects, services, relationships, messages related to this scenario. The scenario is described as:

1. The ELEVATOR reaches a floor.
2. The FLOOR_SENSOR tells the CONTROLLER the arrival of the ELEVATOR.
3. The CONTROLLER turns on the ARRIVAL_LIGHT.
4. The CONTROLLER asks if there is a passenger who wants this floor as a destination.

5. The CONTROLLER asks if there is a summon from this floor. If so, turns the SUMMON_LIGHT off.
6. The CONTROLLER tells the MOTOR to stop.
7. The CONTROLLER tells the ELEVATOR that it's to open the door.
8. The ELEVATOR tells the DOOR to open itself.
9. The DOOR tells the ELEVATOR it its closed again.
10. The ELEVATOR asks the OVERWEIGHT_SENSOR if the ELEVATOR is overweight.
11. The ELEVATOR tells the CONTROLLER it is ready to move now.
12. The CONTROLLER asks the floor of the next summon.
13. The CONTROLLER asks the next destination floor.
14. The CONTROLLER tells the MOTOR to go up or go down.

Figure 5.6 shows the state transition diagram for class CONTROLLER. Figure 5.7 shows the transition trace diagram for the scenario.

As we can see in Figure 5.7, Turn_on, the output event from state $S0$, will invoke ARRIVAL_LIGHT. The invoked method in ARRIVAL_LIGHT is Turn_on_myself. This invocation method is a *one-way invocation*, which means that the invoking object CONTROLLER does not need to pass any value to it and does not require any return value from the event it invoked. On the other hand, after issuing Turn_on event, CONTROLLER will go into state $S1$. The output event from $S1$ If_floor_summoned will invoke Report_on_off_status in SUMMON_LIGHT. Report_on_off_status does not require input value to be invoked. This means that If_floor_summoned can independently invoke SUMMON_LIGHT without the result from CONTROLLER's prior output event, Turn_on ARRIVAL_LIGHT. Therefore, the ARRIVAL_LIGHT and SUMMON_LIGHT can be invoked concurrently.

Continue this analysis with the state transition diagram of the CONTROLLER, the next event from state $S2$ is Turn_off. Because this event occurs upon the input event to $S2$, Report_on_off_status from SUMMON_LIGHT, and Report_on_off_status pass data to CONTROLLER, so SUMMON_LIGHT and SUMMON_BUTTON can't be concurrently invoked. The next event from state $S3$ If_floor_destination does not need any input event to $S3$, and hence it can execute concurrently with the event Turn_off from the prior state $S2$.

After applying the same method to the complete diagram, we can find that the following pairs of objects can be potential invoked concurrently :

(SUMMON_LIGHT, ELEVATOR),
(SUMMON_BUTTON, SUMMON_LIGHT),
(CONTROLLER, SUMMON_LIGHT),

(DESTINATION_LIGHT, SUMMON_LIGHT),
(DESTINATION_LIGHT, SUMMON_BUTTON),
(DESTINATION_LIGHT, CONTROLLER),
(ARRIVAL_LIGHT, SUMMON_LIGHT),
(OVERWEIGHT_SENSOR, CONTROLLER),
(MOTOR, ELEVATOR),
(MOTOR, DESTINATION_LIGHT),
(MOTOR, DESTINATION_BUTTON),
(DOOR, ELEVATOR),
(DOOR, FLOOR_SENSOR),
(DOOR, OVERWEIGHT_SENSOR).

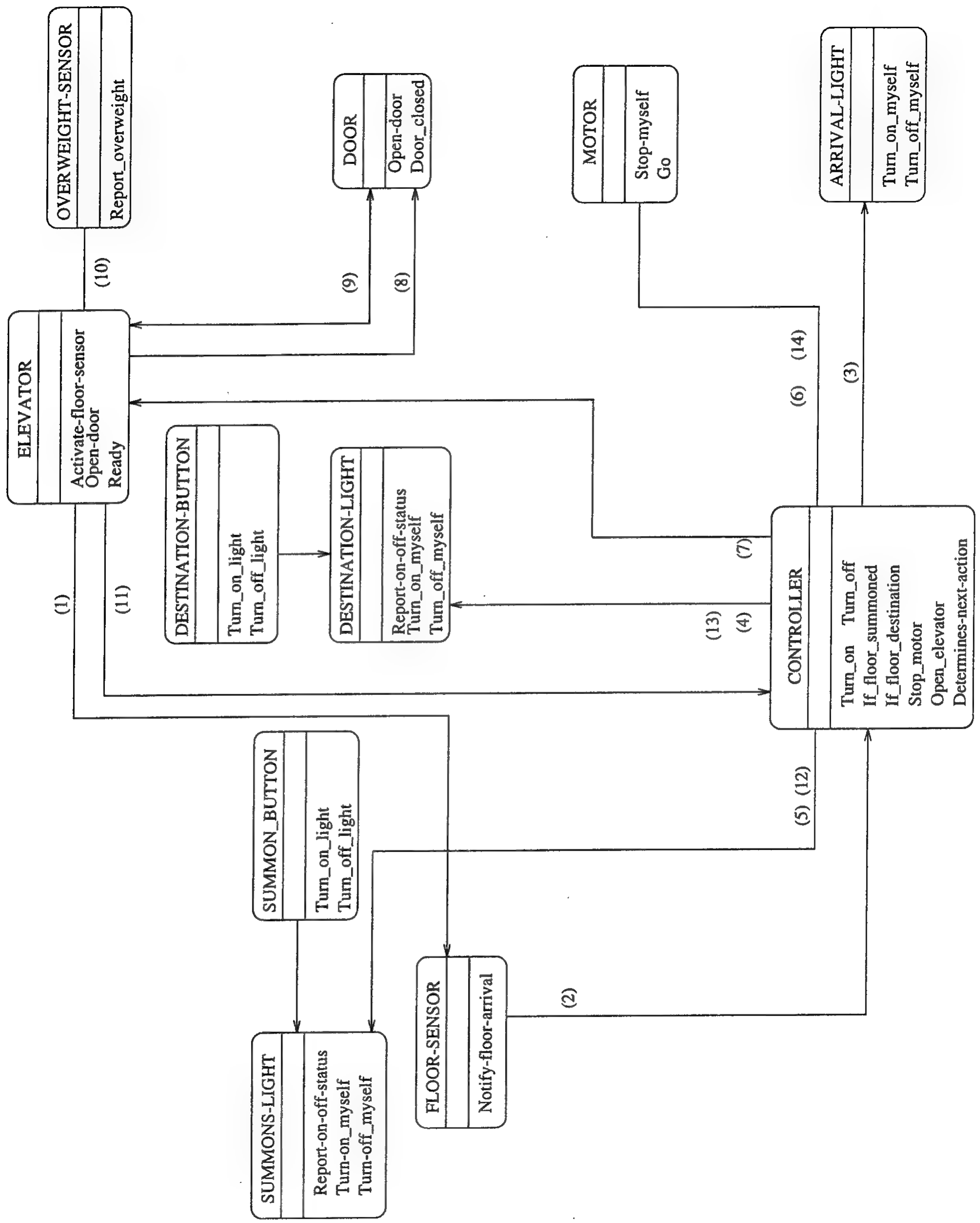


Figure 5.5: An scenario for the elevator system.

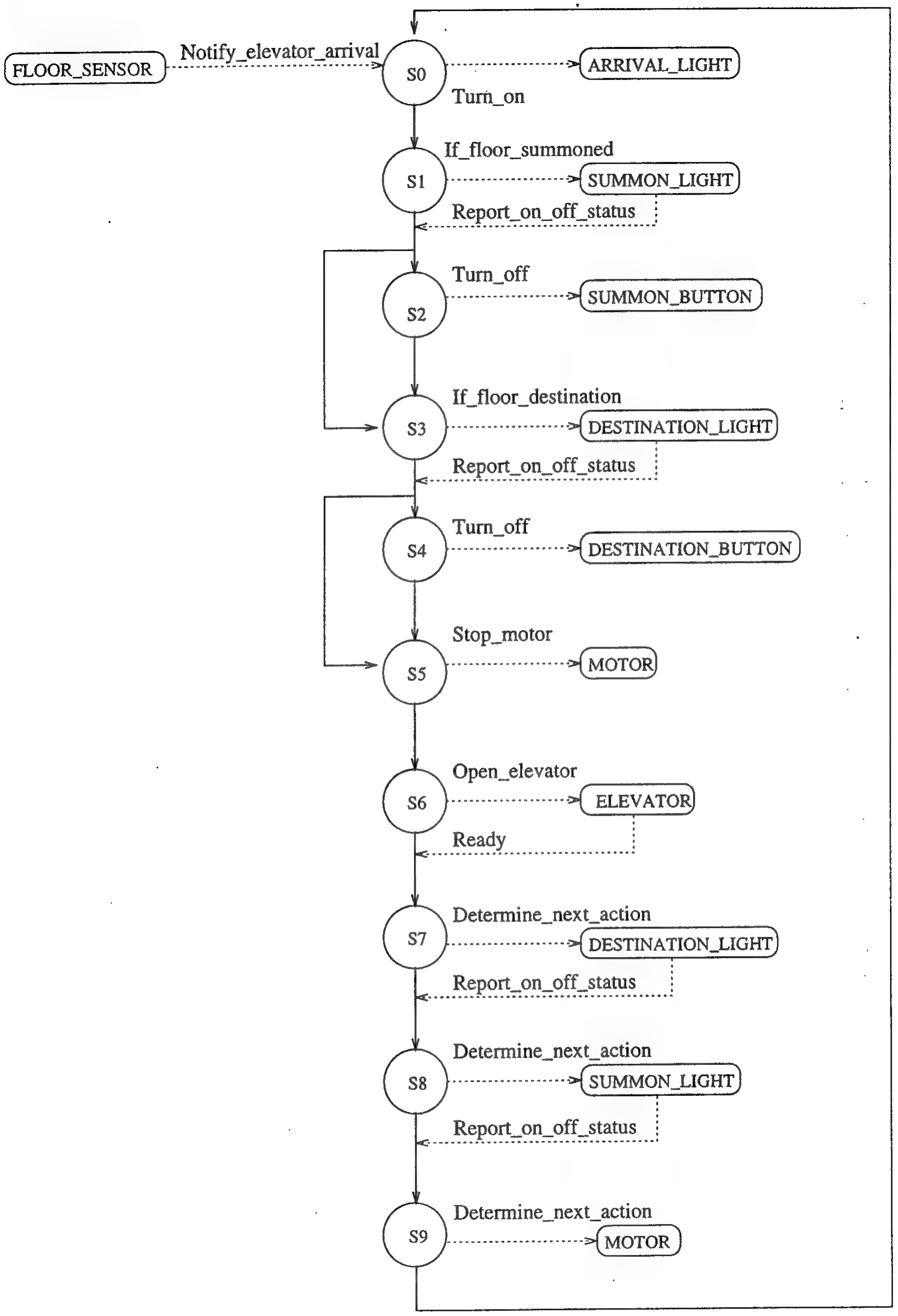


Figure 5.6: The state transition diagram for CONTROLLER of the elevator system.

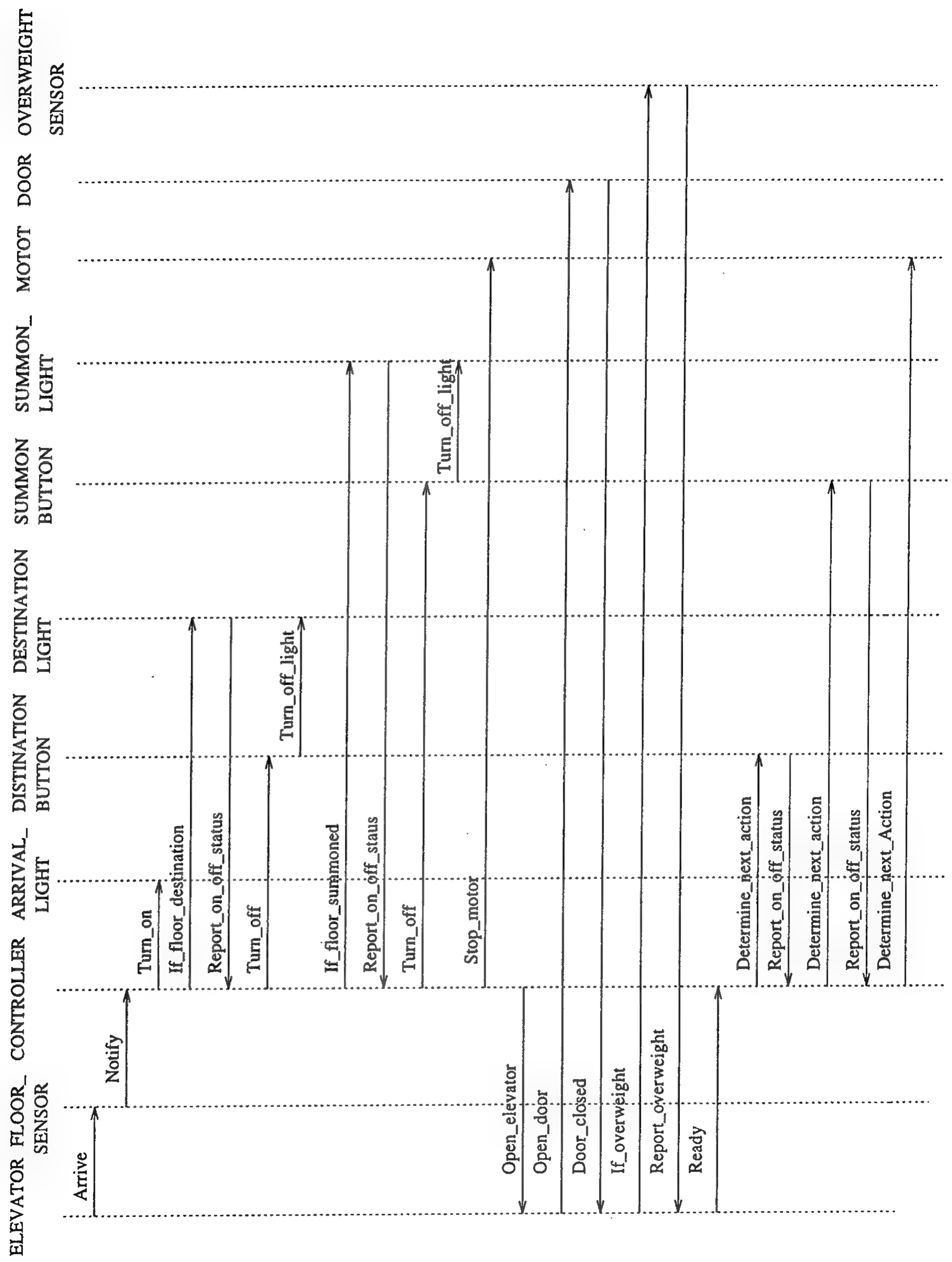


Figure 5.7: The transition trace for a scenario of the elevator system.

Chapter 6

Back-end Translator on A Workstation Cluster

In this section, we will present the back-end translation which translates our superset Intermediate Form 1 (IF1) to PVM/Sun C target language based on the previous work of two other target language translators : nCube C and KSR C [8]. Various IF1 constructs for parallel functions, iterative WHILE loops, IF structures, and common computational operations are identified and translated. The important issues about designing inter-node communications and synchronization are also discussed.

6.1 Target Languages

There are two kinds of MIMD parallel architectures: shared-memory and distributed-memory. In the shared-memory architecture, the processors share a single-addressed memory. In the distributed-memory architecture, each processor has its own memory, cooperative work must be done through explicitly specified inter-node communications and synchronization.

We have developed PROOF/L back-end translators for distributed-memory workstation cluster using the same front-end as two previous back-end translators: one for a distributed-memory parallel machine nCube and the other for a shared-memory parallel machine KSR [8]. Since all these three translators were written in C and the difference mainly exists in the communication and process control, we use similar data structure for method's code generation. Both of workstation cluster and nCube machine use message passing API. Our basic strategy for workstation cluster translator is that starting from the nCube C translator, we establish a communication adapter to simulate nCube message passing API. The process control mode is introduced in section "Clustering and Dynamic Allocation Support". The rest of this section lists all the communication and process control APIs of nCube C and those PVM APIs that are used in workstation translator to establish the adapter under nCube C APIs.

- The nCube C

  The nCube C version 3 [37] consists of a comprehensive set of ordinary C primitives and build-in functions. Several basic primitives for the parallel execution and inter-node communication are:

  - **rexec:** launch an executable program on a subset of processors. It involves allocation of a set of processors, setup of a process table for each processor within the node set, and execution of the program on each processor.
  - **ntest:** a non-blocking way to test existence of messages from other nodes.
  - **nread:** waits for messages and reads them whenever they arrive and satisfy the type format set by the nread. The nread operation is self-blocked. Inappropriate nread operations could lead to deadlocks.
  - **nwrite:** sends messages from one node across the nCube high-speed bus to another one with a type format.

  All these primitives have substantially large communication overheads. There are several other functions used to check the states of processors at run time on the nCube parallel machine:

  - **whoami:** reports a node condition during the run time.
  - **npid:** return current node ID.
  - **ncubesize:** return the hypercube size, which is 2's power.

  The nCube C itself does not provide any primitives to prevent deadlocks or to synchronize physical nodes. Each node basically stands alone itself.

- The PVM/Sun C

  The intended target environment is a cluster of networked workstations. The workstation cluster that we worked on has 8 nodes connected by 100Mb FastEthernet in our laboratory. These nodes are three SunSparc 4, three SunSparc 5, one SunSparc 20, and one SunUltra 1.

  The parallel executing environment on workstation cluster is the PVM (Parallel Virtual Machine) version 3.3.10 [38]. PVM is a software system that permits a heterogeneous collection of Unix computers networked together to be viewed by a user's program as a single parallel computer. PVM transparently handles all message routing, data conversion, and task scheduling across a network of incompatible computer architectures. The system offers a library of standard interface routines that can be called by user's C program. The routines that we used in our translator are :

  - **pvm_spawn:** starts new copies of an executable file task on the virtual machine. Two arguments, "flag" and "where", are used to specify options that control where these new processes be spawn on.
  - **pvm_mytid:** returns the Task ID of this process and can be called multiple times. It enrolls this process into PVM if this is the first PVM call. This Task ID is a global value on the whole workstation cluster.

- **pvm_config:** returns information about the virtual machine, including the number of hosts, different architectures and information of every host: Host ID, name, architecture and relative CPU speed.
- **pvm_initsend:** If the user is using only a single send buffer, then this routine should be called before packing a new message into the buffer. It clears the send buffer and creates a new one for packing a new message. The encoding scheme can be specified to send data between different architecture machines. As all the workstations in our developing environment are Sun-Sparc workstations, the messages at their original format can be understood on any node. So we disable encoding to save marshaling time.
- **pvm_pkbyte,pvm_pkint, pvm_pkfloat, pvm_pkstr:** each of these routines packs an array of the given data type into the active send buffer. They can be called multiple times to pack data into a single message.
- **pvm_send:** labels the message with an integer identifier and sends it immediately to a process specified by a Task ID.
- **pvm_mcast:** just like pvm_send, send a message in active send buffer to several processes specified by a Task ID array.
- **pvm_recv:** is a blocking receive routine. It will wait until a message with specified label has arrived from a specified process.
- **pvm_probe:** If the requested message has not arrived, then pvm_probe() returns 0, otherwise returns receive buffer ID.
- **pvm_upkbyte,pvm_upkint,pvm_upkfloat,pvm_upkstr:** These routines unpack data types from the active receive buffer. The unpacking should be done in the same sequence as the message is packed.

The PROOF/L code is translated into Sun C with extension of PVM function calls. All the generated programs run under PVM runtime daemon.

6.2 Object Cluster Allocation

Objects are clustered to a number of clusters each of which is assigned to and run on a computing node in order to reduce the communication cost among objects. In most cases, the number of object clusters in an application is more than the number of computing nodes. In the workstation version PROOF/L translator, when clusters need to be allocated, a node that has the least load is selected and cluster with the largest number of objects is allocated to it. This procedure is repeated until all clusters are allocated. The procedure will make sure no workstation will be overloaded.

The pseudo code of cluster allocation schema is given below :

```
CLUSTERS    : cluster set to be allocated
Power(WSi)  : computing power ratings of each node
```

```
{\tt for}( each available computing node WSi )
{
LOADi  = get the number of objects running on WSi;
RPOWER = Power(WSi)/1000;
}
{\tt while}( CLUSTERS not empty )
{
Ci = the largest cluster in CLUSTERS;
{\tt for}( each available computing node WSj )
WEIGHTj = (LOADj + Ci)/RPOWERj;
WSk = the node that has the least WEIGHTk;
allocate Ci process onto WSk;
LOADk = LOADk + sizeof(Ci);
CLUSTERS = CLUSTERS - Ci;
}
```

Power(WSi) is an approximate estimate of their relative computing power. Define the slowest machine in the computing network with power 1000 and compare all the others relative to it. In our workstation cluster,

```
Power(SunSparc5) = 1000 Power(SunSparc4) = 1100
Power(SunSparc20) = 2000 Power(SunUltra1) = 2200
```

6.3 Target Languages Code Generation

IF1 (Intermediate Form 1) is used to have graphical representations for PROOF/L programs.

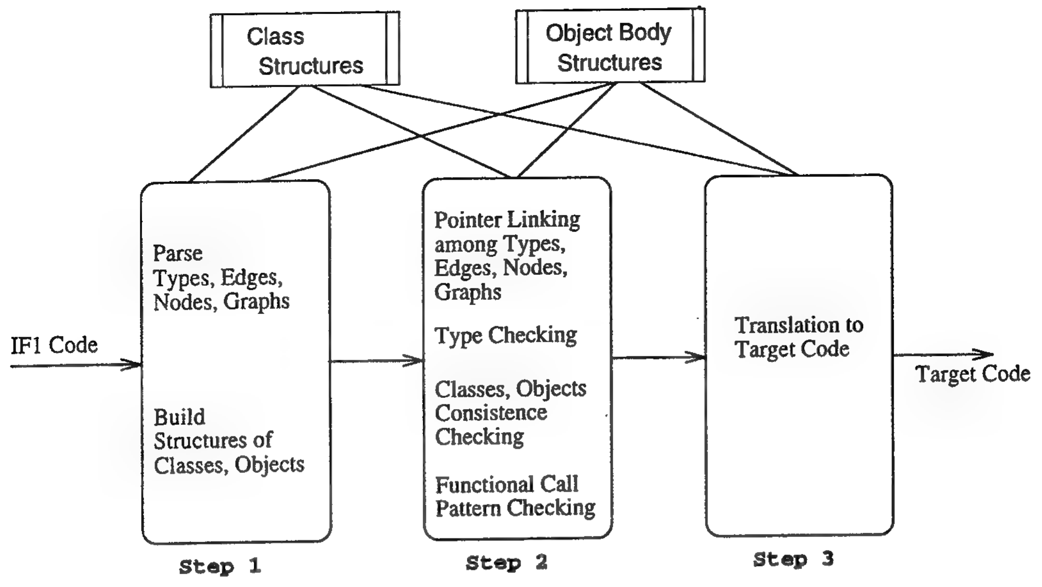
The translation from IF1 to target languages consists of three steps: parse IF1 code, structural linking and translation, as shown in Figure 6.1.

6.3.1 Parse IF1 Code

IF1 code consists of types, edges, nodes, graphs, and numerical relations among them. Because there are no explicit mechanisms to describe object-oriented concepts – classes and objects – in the original IF1 syntax, extensions have been made for the IF1 code to keep the class and object information for the back-end translation. All the class and object information is stored in the IF1 type headers, graph headers, and object headers. The following IF1 code describes these constructs.

1. The class's composition:

```
T      <Type_id>      RECORD<next>   %na =<class_name>
```



Class Structures
Object Body Structures
IF1 Code
Parse Types, Edges, Nodes, Graphs
Build Structures of Classes, Objects
Pointer Linking among Types, Edges, Nodes, Graphs
Type Checking
Classes, Objects Consistence Checking
Functional Call Pattern Checking
Translation to Target Code
Target Code
Step 1
Step 2
Step 3

Figure 6.1: The translation steps from IF1 to a target code.

2. The class's method header:

   G   <Type_id>   <Class_Name.Method_Name>

3. The object body part:

   X   0   <Class_Name.Object_Name>

The comment fields are only used for reference and ignored during the translation.

The execution sequence within a graph is sorted by detecting the dependency among the nodes, edges, and their numerical linkages in the graph. The algorithm below demonstrates the sorting of execution sequence:

```
seq = 0;
mark all the nodes unsorted;
for (each node) {
  for (each input edge) {
     if (each input edge is literal ||
              not an output edge from an unsorted  node) {
        continue;
     } else {
       break;
     }
  }
  if (all the input edges are checked) {
    set order of current node = seq;
    seq++;
  }
```

```
    }
    if (any unsorted nodes left) {
      report error;
      exit(-1);
    }
```

After *types*, *edges*, *nodes*, and *graphs* of the entire IF1 code have been scanned, class and object structures can be derived through implicit class and object information in the new nodes of the IF1 code. The information includes class compositions in *types*, method names in *graphs* and object names in *graphs*.

6.3.2 Structural Linking

After numerical relations among types, edges, nodes, and graphs have been identified and built in the data structure during the first step, all these numerical relations are converted to pointer linking between types and edges, edges and nodes, types and graphs, and nodes and graphs. Method calls include four types:

- Built-in functional calls (denoted as imported functions in the IF1 code). They include `append_left`, `append_right`, `tail`, `head`, `last`, `inc`, `dec`, `null`, `delay`, `while`, etc.
- Global functions. The class GLOBAL is a class without any data structure but methods. It is a collection for public methods, in which every method can be used by any other classes or object bodies without any difference comparing to using their own methods.
- Method calls within a class. Currently a class method can only call another method within the scope of the same class, besides build-in and global functions.
- Method calls within an object body. Objects can call any methods available within the entire scope.

In addition, GUARD structures within methods are detected for each class. It is the only way for different objects in the PROOF/L code to synchronize one another. These structures are represented as GUARD compound nodes. The text representation is shown as follows:

```
{
G       0       GUARD 1 (structure)
C The predicate needed to be realized to continue the execution
...
G       0       GUARD 2 (structure)
C The body executed after the guard predicate above becomes true
...
} <node_id>     GUARD    2    1    2
```

The number of input edges to computational nodes are verified, and edges for input arguments to method CALL nodes are checked against method prototypes. Type consistency checking is also applied to input and output data flows, which are represented by edges, among all the simple nodes in the IF1 code.

PROOF/L parallel structures will be detected in two ways at the IF1 level:

- Detect `alpha` (apply to all) and `beta` (distributed apply) parallel functions through data dependency among CALL and LBUILD nodes.
- Find a built-in parallel function call named `delta` (data partition) in the IF1 code.

6.3.3 Unique Data Structures

Since all the three target languages are C-extension, we use the same data structure for the PROOF/L translators. We will mainly use PVM/Sun C to explain the steps of translation. When differences between PVM/Sun C and nCube C or KSR C exist, further discussion will be given.

In order to realize the functional features in the PROOF/L language, only two kinds of data formats have been used: ATOM and SEXP. They are shown in Figure 6.2.

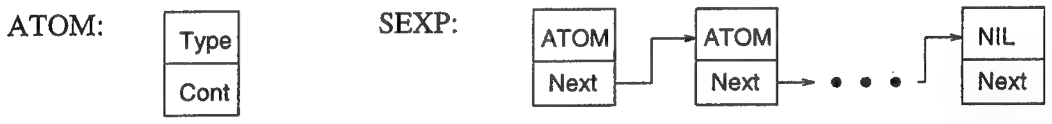


Figure 6.2: The data format of the translated target C code

Concatenated structures for PROOF/L class compositions are represented by using SEXPs. An initialization function for each class will be provided to initialize all components within the correspondent class composition: 0 for the integer or float, FALSE for the boolean, "(empty string) for the string, NIL for the list. So there are no explicit data definitions of classes at the C code level.

The main feature of the PROOF language is to combine the functional and object-oriented domains together. In order to save the class and object information for the back-end translation, we extend the IF1 to carry the class and object information across PROOF/L code to target C code. We apply the data structures, ATOM and SEXP, to realize all the functional features. Furthermore, lists used in the PROOF/L are a type of heterogeneous lists, which is similar to those in LISP. A number of list manipulation functions have been given, such as List Constructs [](square brackets), `append_left, head, tail, append_right, null?`, etc.

All other data types in the PROOF/L – integer, float, boolean, and string – are implemented with single type, called ATOM, with the unique type code embedded inside. The binary, boolean, relational and unary operations are applied in the following way:

```
        verify type of the first atom depending on the operation;
        extract content of the first atom;
        verify type of the second atom depending on the operation if
        applicable;
        extract content of the second atom if applicable;
        apply the operation to content(s) of the atom(s);
        compose a new atom with the result of the operation and
        appropriate type;
```

The underlining unique interface for processing different data types gives considerable flexibility to programmers, but sacrifices execution performance.

A functional language always involves recursion and PROOF/L is no exception. Recursive function calls not only are resource-consuming, but also limit computational capacity. Currently, there is no implicit recursion removal during the translation. An iterative functional structure, While loop, has been used to avoid these explicit recursive calls. Because of the specialty of While loops, a library routine has been written to realize iterations of While loops.

```
PROOF/L format:
        while(<predicate lambda exps>, <body lambda exps>)
<an input to lambda>

Translated C format:
        result = <an input to lambda>;
        while (1) {
        if ( <predicate lambda exps> ( result ) ) {
                result = ( <body lambda exps> ( result );
        } else {
                break;
        }
        return result;
```

Our own PROOF/L library routines for supporting functional operations have been provided. They include all the built-in functions (except parallel `delta` function), copy routines, and garbage collections routines. Besides all these routines, packet assembling and disassembling routines have been written for message-passing type communications in all the three target languages, which will be described later.

The broad translation for a class is described as follows:

```
IF1                                     PVM/Sun C
C Class <name>                          /* Class <name> begins */

C Class Composition                     /* Class composition */
...
```

```
C Class Methods                          /* Methods <Class_name.Method_
                                                                name 1>
                                            begins */
G <Type_id>  <Class_name.Method_name 1>  void <Class_name.Method_name 1>
                                              (<I/O Type>)  {
...                                      ...
                                         }  /* End of method */
...                                      ...

C Class Methods                          /* Method <Class_name.Method_
                                                               name k>
                                            beings */
G  <Type_id> <Class_name.Method_name k>  void <Class_name.Method_name k>
                                              (<I/O Type>)  {
...                                      ...
                                         }  /* End of method */
end Class                                /* Class ends <name> */

C Extra procedures for every class       /* class-method lookup table
                                            begins */
                                         void <name>_func_dispatcher(){
                                         /* Lookup Table */
                                         ...
                                         }
                                         /* class-method lookup table
                                            ends */
```

The broad translation for an object is described as follows:

```
IF1                                      PVM/Sun C
                                         /* Declaration */
X  0  <Class_name.Object_name>           SExp *<object_namename>;
C nodes(Simple or Compound), edges       /* Object <Class_name.Object_
                                            name> begins */
...
                                         void <Class_name_Object_name>()
                                         { ... }
```

6.3.4 Clustering and Dynamic Allocation Support

In PROOF/L, any application is a group of objects. Every object is an independent process. Some objects are allocated on the same node because clustering restriction or work load balancing while the others may be on different nodes. They all run the same program and this program will determine which object it should act as. In the runtime model of the translators for NCube and KSR, there is no starting object. Every CPU has a physical serial number which is used to determine the which objects should be run on it. In order to assign processes to objects on a workstation cluster, a starting object is generated automatically by the translator besides the PROOF/L program. The activity of this starting object is shown in Figure 6.3. All the objects that are defined in PROOF/L program are spawn by the starting object. Then, an array that contents all these PVM process IDs is broadcasted from starting object. After receiving this array, each process can determine the index number in this array which is used as the number of objects, and send back ACKnowledge message to the starting object. When all the ACK is returned, the starting object is terminated, so no more overhead is added to the computing network while those objects start working.

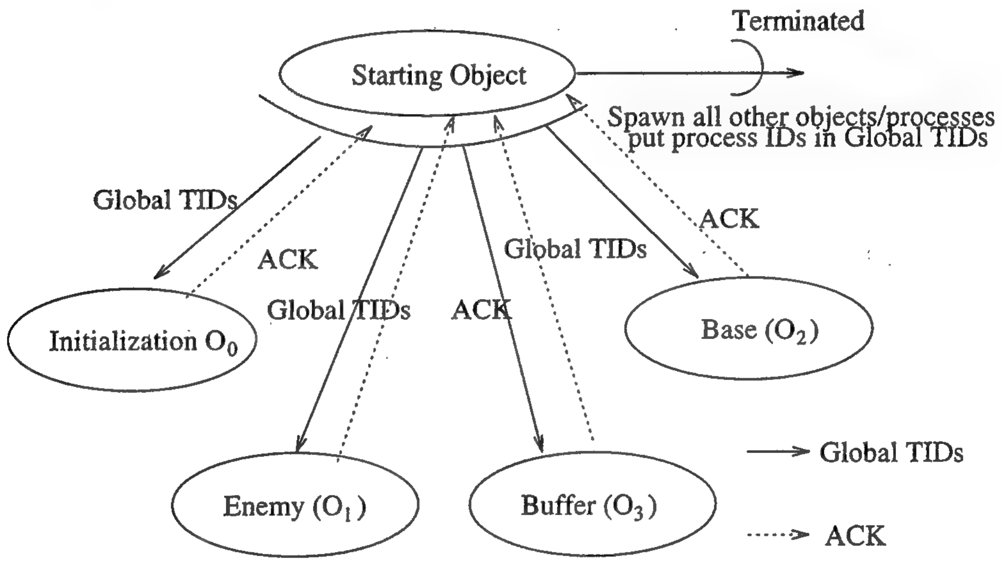


Figure 6.3: The activity of the starting object

The program initialization method is included in Object O_0. All the methods in both active and passive objects can be invoked by this initialization method. Before O_0 terminates, a synchronization message will be broadcasted to the rest objects.

Upon receiving the synchronization message, the objects start to execute their body methods. For active objects, their bodies are defined by user's program. They will not offer services (allow its methods be called) to other objects anymore. For passive objects, their bodies are defined by the translator as a service loop. Its function is to receive function call requests, call the corresponding methods and send the results back.

This workstation cluster back-end translator supports all the previous PROOF/L features. It also supports object clustering and dynamic allocation.

From the our OOA/OOD CASE tool, the user can get the clustering information in the format of a file with object name list each line for each cluster. When the translator compile PROOF/L source code, it also reads in this clustering information and translates the object names to internal object ID array. This array will be defined as constant in target language code and will be used by the running-time object allocation algorithm described in previous section.

In order to run the allocation dynamically, this function code has to be executed at the system initiating period. The starting object we add into the system satisfies this requirement. The advantage of using starting object as allocation host is that the allocation function code can directly be pre-programmed and no compiling time generation is needed. This makes the allocation code more neat and efficient.

6.3.5 Output File Organization

In order to modularize the entire translated target C code, four basic files are generated after the translation:

- **class.h** contains all the object declarations and all the necessary C "include" files. All class method prototypes and class initialization function prototypes are also listed here. This is the main header file for the entire translated C code.
- **methods.c** contains all the class methods, bounded by comment marks for each class. It also contains class-method lookup tables for the purpose of communications among different objects.
- **objects.c** contains bodies of all the objects.
- **main.c** provides the initialization of all the available physical nodes, and associate each object body with a process in PVM (a single node in the nCube C and a thread for each object body in the KSR C), dispatch all the corresponding controls to object bodies, synchronize all the objects to start execution simultaneously, and finally do the cleanup when all the objects are terminated.

  The main.c is given as follows:

```
initialize all the objects;
initialize all threads or nodes;

switch(<Thread ID>) {
        case 1: <Object1_func>();
        break;
        case 2: <Object2_func>();
        break;
        case 3: <Object3_func>();
        break;
        case 4: <Object4_func>();
        break;
        case 5: <Object5_func>();
        break;
}
synchronize all threads to start at the same time;

wait for terminations of all threads.
```

All the classes and objects are translated based upon the structures shown above, and corresponding portions of code are put into the header file or different .c files to modularize the problem. The entire set of .c and .h files will be put into a directory according to what were provided. Also a general makefile for the purpose of translation of target C code is given.

6.3.6 Distributed Method Invoke and Building Parallel Functions

As mentioned in Section 6.1, distributed-memory parallel machines, like the workstation cluster or nCube, need to explicitly specify communications among different physically separated nodes. On the other hand, shared-memory parallel machines, like the KSR, provide communications among different nodes at the kernel level, which releases this task from the programmers. Our translators for workstation cluster and nCube emulate communications for the shared-memory KSR machine in order to provide the unique structures to translate the IF1 code for three different target languages with as few variations as possible.

All the concurrent objects in PROOF/L that can be executed in parallel are dispatched to different threads or nodes. Each object is a computation unit of its own. The guard statements in methods are used to handle synchronization among PROOF/L objects executed on different nodes or threads.

Periodic communications among different threads or nodes are made under two conditions: 1) call methods of other objects, and 2) applications of the parallel functions: `alpha`, `beta`, and `delta`.

Condition 1) is for the purpose of method invocations between two different objects. The reasons for one object to invoke another object's method are query for the state information of another object, and change of the state of another object through the reception function (this is the only way to modify the state of an object).

The scheme for this kind of communication is shown in Figure 6.4. A request object sends the method id to indicate which method it wants to call to a responder object, assembles all the arguments necessary to that method, except state information of the responder object, into a stream packet, and then sends the packet through the network. After the method id received, the responder object dispatches the stream packet of arguments to the method that was requested. Arguments get extracted from the stream packet and passed to the method associated with the state information of the responder object. The final outputs of the method are assembled again into a stream packet and passed back to the request object which in turn will disassemble the returned packet to get the results it expects. Overheads for assembling and disassembling are necessary for providing a unique and simple interaction between two different objects, and they are much less time-consuming than communication overheads across two different processors. Another reason for us to assemble all the arguments together and send once across two nodes is that multiple communications with small packets are more time-consuming than a single communication with a large packet.

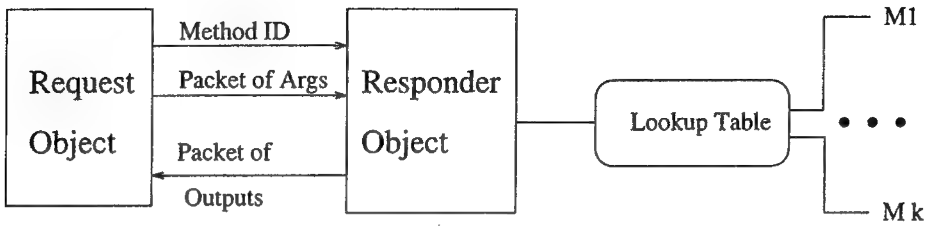


Figure 6.4: The underlying PROOF/L communication scheme

The code skeleton for a class-method lookup table is shown as follows:

```
/* message comes in */
switch (method_id) {
case 1:

method 1 set up;        /* receive argument packets and
                           disassemble */
Call method 1;
method 1 feedback;      /* assemble final results */

break;

case 2:
...
}
```

Class-method lookup tables are required for the distributed-memory nCube machine. On the other hand, an object on the shared-memory KSR machine does not need to explicitly pass messages over to other objects because they can call another object's method directly due to the fact that all objects shared their state information with others, while objects on the nCube machine own their state information themselves. But, the sequence for calling methods is still the same, including assembling and disassembling arguments to methods.

Deadlock is entirely avoided. Each object controls its own thread by executing its body. The loose-coupled relations among objects are well maintained by all the GUARD statements within methods of each object. Objects are ready to serve other objects' requests when they enter unsatisfied GUARD statements.

Bottlenecks in the PROOF/L for a program normally are writable objects. When many objects want to invoke methods of writable objects, the program's execution pace slow down considerably.

Condition 2) is to execute a single function with multiple ranges of data in parallel (`alpha`, apply to all or `delta`, data partition) or multiple functions with their own data ranges in parallel (`beta`, distribute apply). The translator for workstation cluster can execute all the built-in functions and all the global functions in parallel.

Same reasons as the condition 1) are applied here for our assembling arguments to methods before calls and disassembling stream packets to extract results after methods finish executions.

- `alpha` function

```
PROOF/L format: alpha <method> ([args 1], [args 2], ...)
target C format:        assemble args 1;
                        launch a process to call method;
                        assemble args 2;
                        launch a process to call method;
                        ...
                        wait for return packet 1;
                        disassemble packet 1;
                        wait for return packet 2;
                        disassemble packet 2;
                        ...
                        build all return results into a list.
```

- `beta` function

```
PROOF/L format:         beta (method 1, method 2, ...) ([args 1],
                                                        [args 2], ...)
target C format:        assemble args 1;
                        launch a process to call method 1;
                        assemble args 2;
                        launch a process to call method 2;
                        ...
                        wait for return packet 1;
                        disassemble packet 1;
                        wait for return packet 2;
                        disassemble packet 2;
                        ...
                        build all return results into a list.
```

- `delta` function

```
PROOF/L format:         delta <method> ([low_bound], [upp_bound],
                                       <rest args> ...)
target C format:        assemble all args: low_bound, upp_bound,
                                        rest args ...;
get N=the number of available computing nodes
                        launch N method processes;

                        /* Depending on physical capacity,
                           launch processes as many as possible */
                        ...
                        wait for return packet 1;
                        disassemble packet 1;
                        wait for return packet 2;
                        disassemble packet;
                        ...
                        build all return results into a list.
                            /* Dimension undermined */
```

Chapter 7

Examples

In this chapter, we present two examples for illustrating our approach. The first example is an ATM system adopted from [12], and the second example is the Air Force Defense System used in [8].

7.1 The ATM System Example

We will use a hypothetical Automated Tellar Machine (ATM) example to demonstrate our approach involving both coarse grain and fine grain parallelism. In this example, we focus on both communication and computation aspects.

7.1.1 Specifications of A Hypothetical ATM System

Supposed that we are requested to develop an ATM system, whose requirement specification is as follows:

> Design a system to support a computerized banking network including both human cashiers and automatic teller machines(ATMs) to be shared by a consortium of banks. Each bank provides its own computer to maintain its own accounts and process transactions against them. Cashier stations are owned by individual banks and communicate directly with their own bank's computers. Human cashiers enter account and transaction data. Automatic teller machines communicate with a central computer which clears transactions with the appropriate banks. An automatic teller machine accepts a cash card, interacts with the user, communicates with the central system to carry out the transaction, dispenses cash, and prints the receipt. In order to ensure the security of data exchanged between the ATMs and the central computer, data must be encrypted before being sent out and decrypted after

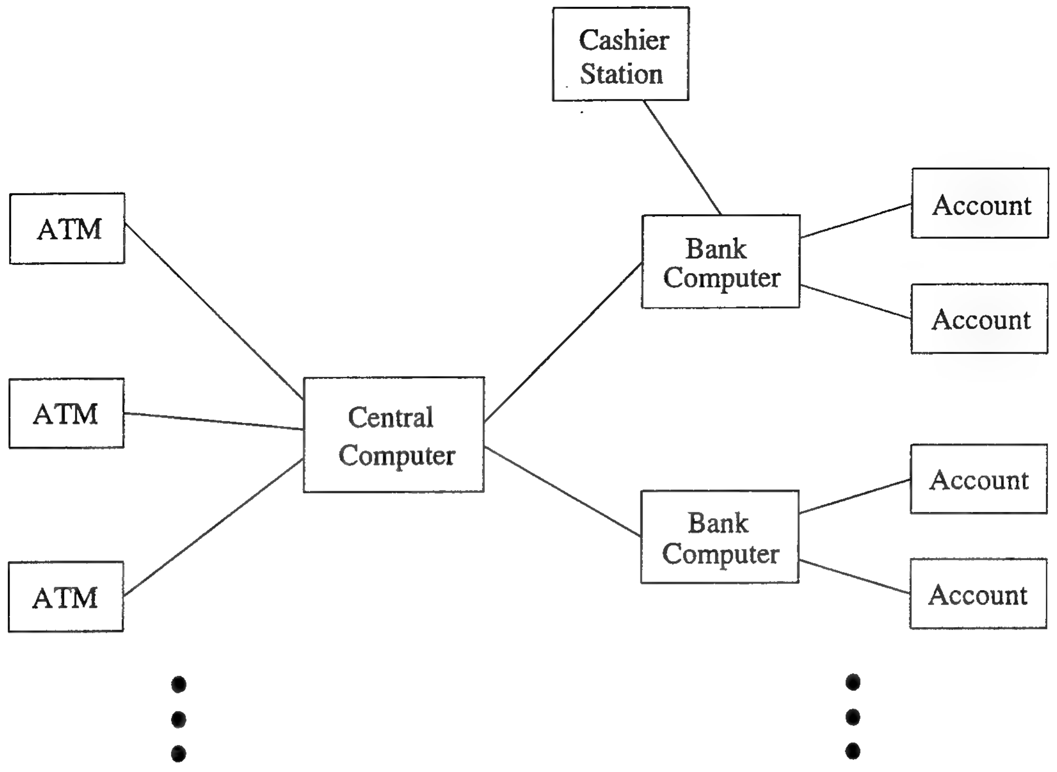


Figure 7.1: An ATM system.

being received. The system must handle concurrent accesses to the same account correctly. The banks will manage their own software for their own computers. We are requested to design the configuration and communication software for the ATM and the central computer as shown in Figure 7.1.

7.1.2 Object-Oriented Analysis

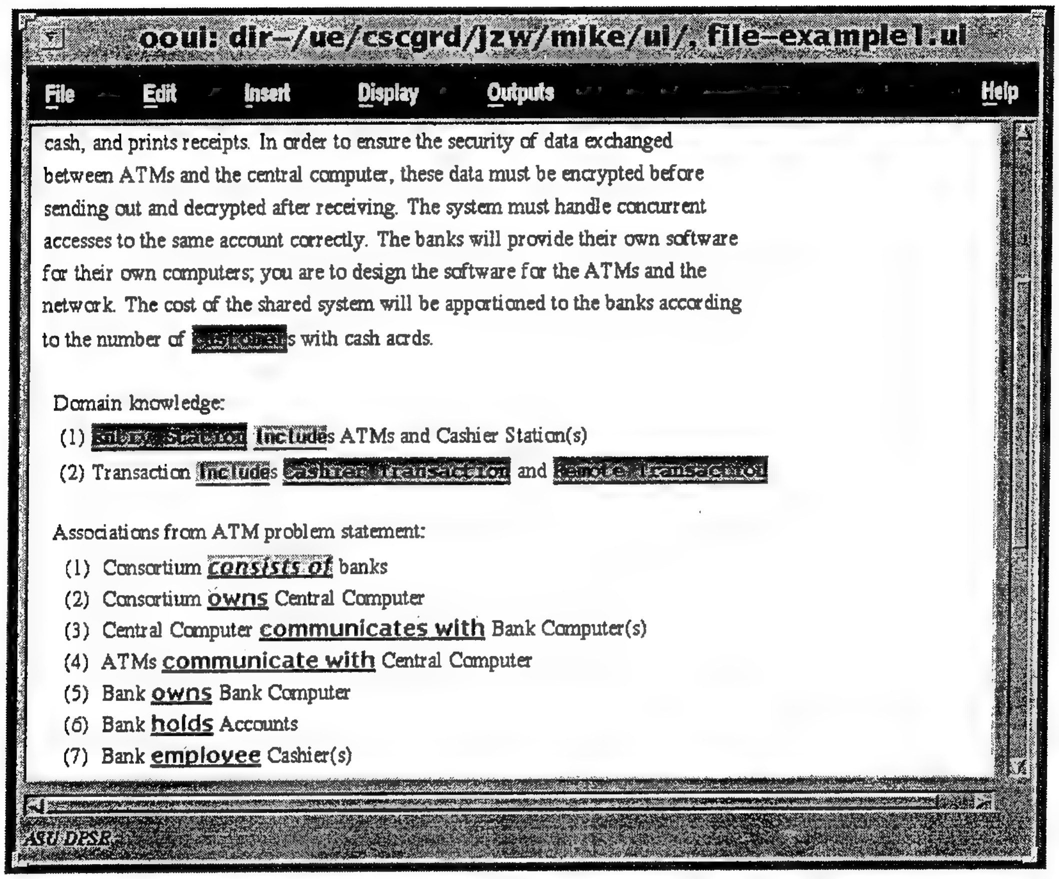
1. Identifying classes and objects

   Our object-oriented analysis tool can help software designer analyse problem statement. A software designer can use the tool to identify classes/objects and their interfaces. Figure 7.2 is a snapshot of object-oriented analysis tool.

   Using our object-oriented analysis tool, we identify the following classes from the requirement specification of the example:

   - Account — A single account in a bank against which transactions can be applied. Accounts may be of at least two types, checking and savings. A customer can hold more than one account.

     ```
     attribute: account code
                balance
                credit limit
     ```



cash, and prints receipts. In order to ensure the security of data exchanged between ATMs and the central computer, these data must be encrypted before sending out and decrypted after receiving. The system must handle concurrent accesses to the same account correctly. The banks will provide their own software for their own computers; you are to design the software for the ATMs and the network. The cost of the shared system will be apportioned to the banks according to the number of customers with cash acrds.

Domain knowledge:

(1) Entry Station includes ATMs and Cashier Station(s)
(2) Transaction includes Cashier Transaction and Remote Transaction

Associations from ATM problem statement:

(1) Consortium consists of banks
(2) Consortium owns Central Computer
(3) Central Computer communicates with Bank Computer(s)
(4) ATMs communicate with Central Computer
(5) Bank owns Bank Computer
(6) Bank holds Accounts
(7) Bank employee Cashier(s)

Figure 7.2: The object analysis result of the ATM system example using our OOA tool.

```
                    password
                    type
```

- ATM — A station that allows customers to enter their own transactions using cash cards as identification. The ATM interacts with the customer to gather transaction information, sends the transaction information to the central computer for validation and processing, and dispenses cash to the user. We assume that an ATM is always connected to the network system.

```
attribute: station code
           cash available on this station
           dispensed
```

- Bank — A financial institution that holds accounts for customers and that issues cash cards authorizing access to accounts over the ATM network.

```
attribute: bank code
           name
```

- Consortium — An organization of banks that commissions and operates the ATM network. The network only handles transactions for banks in the consortium.

- Bank computer — The computer owned by a bank that interfaces with the ATM network and the bank's own cashier stations. A bank may actually have its own internal network of computers to process accounts, but we are only concerned with the one that connects to the network system.

```
attribute: bank code
```

- Cash card — A card assigned to a bank customer that authorizes access of accounts using an ATM machine. Each card contains a bank code and a card number, most likely coded in accordance with national standards on credit cards and cash cards. The bank code uniquely identifies the bank within the consortium. The card number determines the account that the card can access. A card does not necessarily access all of a customer's accounts. Each cash card is owned by a single customer family, but multiple copies of it may exist, so the possibility of simultaneous use of the same card from different machines must be considered.

```
attribute: card code
```

- Cashier — An employee of a bank who is authorized to enter transactions into cashier stations, accept cash and checks from customers, and dispense cash to customers. Transactions, cash, and checks handled by each cashier must be logged and properly accounted for.

```
attribute: employee code
           name
```

- Cashier station — A station on which cashiers enter transactions for customers. Cashiers dispense and accept cash and checks; the station prints receipts. The cashier station communicates with the bank computer to validate and process the transactions.

  ```
  attribute: station code
  ```

- Cashier Transaction — A subclass of 'Transaction'. It is entered by cashier through Cashier Station.

- Central computer — A computer operated by the consortium which dispatches transactions between the ATMs and the bank computers. The central computer validates bank codes, but does not process transactions directly.

- Customer — The holder of one or more accounts in a bank. A customer can consist of one or more persons or corporations. The same person holding an account at a different bank is considered a different customer.

  ```
  attribute: name
             address
  ```

- Entry Station — An abstract class. It includes ATMs and Cashier Station(s).

  ```
  attribute: kind
  ```

- Remote Transaction — A subclass of 'Transaction'. It is entered through ATM.

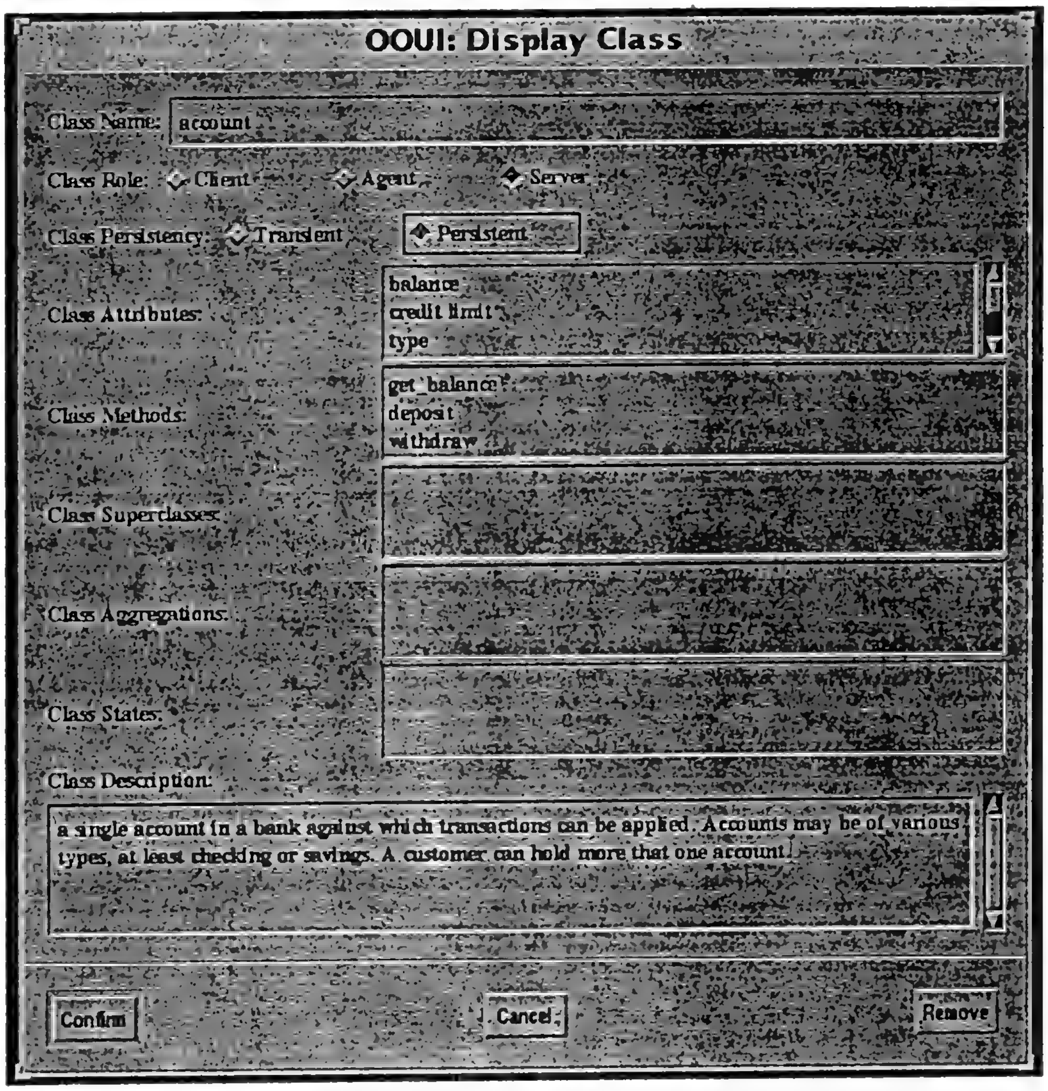
  Transaction — A single request for operations on one account of a single customer. The different operations must balance properly.

  ```
  attribute: kind
             date-time
             amount
  ```

As in all of these classes, those we are most concerned with are account, central computer, bank computer, customer, and account, so we identify the following objects:

A — corresponding to a single ATM

CC — corresponding to the central computer



Class Name: account

Class Role: Client Agent [✓ Server]

Class Persistency: Transient [✓ Persistent]

Class Attributes:
balance
credit limit
type

Class Methods:
get_balance
deposit
withdraw

Class Superclasses:

Class Aggregations:

Class States:

Class Description:
a single account in a bank against which transactions can be applied. Accounts may be of various types, at least checking or savings. A customer can hold more that one account.

Figure 7.3: The interface for designing a class using the OOA tool.

BC — corresponding to a single bank computer

CU — corresponding to a single customer

AC — corresponding to a single account

2. Determing class interface

   Our object-oriented analysis tool also can help software designer design the interface of a class. A software designer can specify the role, attributes, methods, and other properties of a class. Figure 7.3 is the user interface of designing a class in our tool.

   Class interface of an object consists of the input and output parameters and their types. The class interfaces of various objects identified in the previous subsection are given as follows:

```
class account
{ attribute : account code
              balance
              credit limit
              password
              type
  method   get_balance( Account -> Account' x amount )
  method   deposit( Account x amount -> Account' x amount')
  method   withdraw( Account x amount -> Account' x amount')
}

class ATM
{ attribute : station code
              cash on hand
              dispensed
  method display_main_screen( ATM -> ATM' )
  method apply_a_publickey( ATM -> ATM' )
  method request_password( ATM x card -> ATM' )
  method encrypt( ATM x data -> ATM' x message )
  method decrypt( ATM x message -> ATM' x data )
  method verify_account( ATM x message -> ATM'  )
  method request_transaction_kind( ATM -> ATM' )
  method request_transaction_amount( ATM x kind -> ATM' )
  method send_a_transaction( ATM x message -> ATM' x message' )
  method dispense_cash( ATM x transaction_success -> ATM' x cash )
  method request_take_cash( ATM -> ATM' )
  method request_continuation( ATM x take_cash -> ATM' )
  method print_receipt( ATM x terminate -> ATM' )
  method eject_card( ATM -> ATM' x card )
  method request_take_card( ATM -> ATM' )
}

class central_computer
{ method produce_a_public_key( CC x request_a_public -> CC' x public_key )
  method send_a_public_key( CC x public_key -> CC')
  method encrypt( CC x data -> CC' x message )
  method decrypt( CC x message -> CC' x data )
  method verify_card_with_bank( CC x data -> CC' x account_OK )
  method send_ATM_acoount_OK( CC x account_OK -> CC' )
  method send_a_transaction_to_bank( CC x data -> CC' x transaction_OK )
  method send_ATM_transaction_OK( CC x transaction_OK -> CC')
}

class bank_computer
{ attribute : bank code
```

Table 7.1: Object classification of the ATM system example.

Classification	Objects
Active	CU, A
Passive	AC
Pseudo-active	CC, BC

```
  method verify_bank_account( BC x data-> BC' x account_OK)
  method send_central_computer_account_OK( BC x account_OK -> BC' )
  method process_transaction( BC x data -> BC' x transaction_OK )
  method send_central_computer_transaction_OK( BC x transaction_OK -> BC' )
}

class customer
{ attribute : name
              address
  method insert_card( CU -> CU' )
  method enter_password( CU -> CU' x password  )
  method enter_transaction_kind( CU -> CU' x kind  )
  method enter_transaction_amount( CU -> CU' x amount )
  method take_cash( CU -> CU' )
  method terminate( CU -> CU' )
  method take_card( CU -> CU' )
}
```

3. Specifying Dependency and Communication Relationships Among Objects

   Once the class interfaces are obtained for all the classes, we can establish the dependency and communication relationships among the objects from the object-oriented analysis phase. Figure 7.4 gives the dependency and communication relationships among these objects.

4. Identifying Active, Passive and Pseudo-Active Objects

   From the requirements specification and from the object communication diagram shown in Figure 7.4, we identified *CU*, *A* as active objects, *CC*, *BC* as pseudo-active obejects, *AC* as passive object. These objects as shown in Table 7.1.

5. Identifying Shared Objects

   From the object communication diagram as well as the object behavior, we identify the object *CC* as a shared object.

6. Checking for Completeness and Consistency of the Object-Oriented Analysis

   By tracing through the behavior of the objects and looking at the class interfaces, we can see that the object-oriented analysis is complete and consistent.

   After building the class and object hierarchy, we can apply the object clustering algorithm to analyze the coarse grain parallelism among the objects.

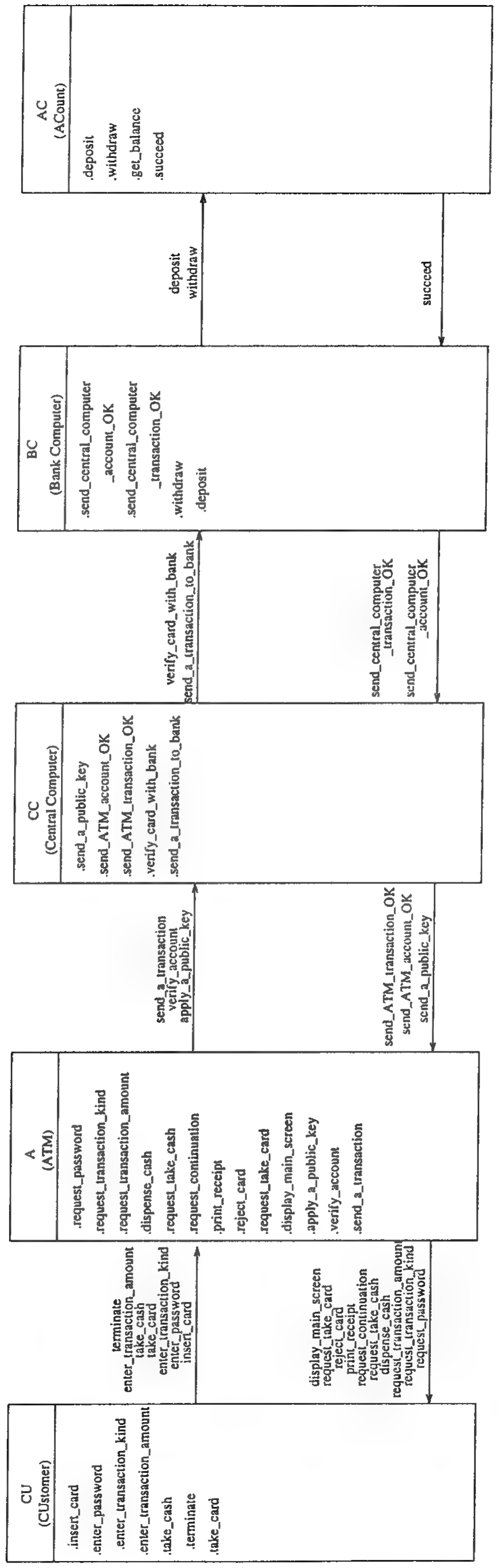


Figure 7.4: The object communication diagram for the set of decomposed objects of the ATM system example.

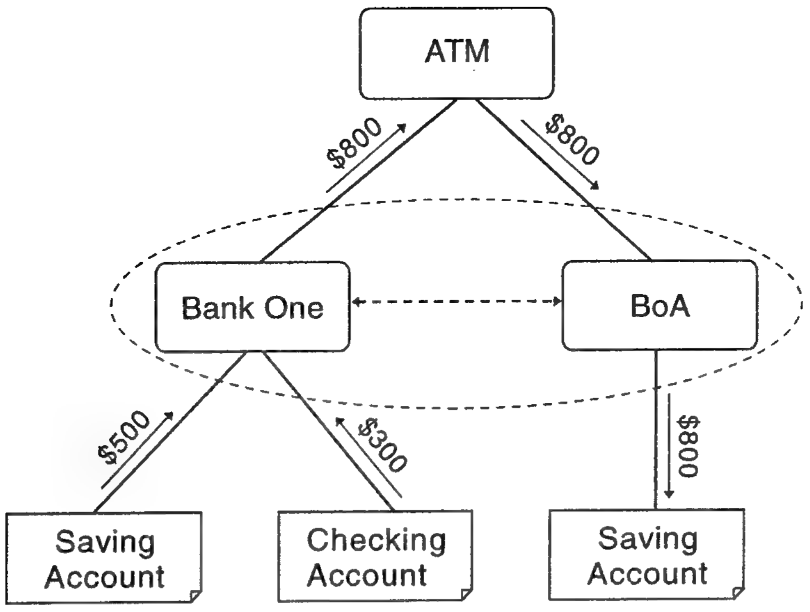


Figure 7.5: The transaction diagram for a transfer between two banks.

7.1.3 Object Clustering

Assume that a customer owns two accounts — account1 (saving account) and account2 (checking account) — in Bank One and one account — account3 (saving account) — in Bank of America. Now he wants to transfer $500 from account1, $300 from account2 to account3. Figure 7.5 gives its transaction digram.

Figure 7.6 is the object structure before applying the clustering algorithm. The following is the clustering result from our tool.

```
Cluster #1:
  "account1"
  "ATM-Rural"
  "Bank One"
  "Central Computer"
  "Bank One Conputer"
Cluster #2:
  "account2"
  "Bank of America"
  "Consortium"
Cluster #3:
  "account3"
  "Bank of America Computer"
Cluster #4:
```

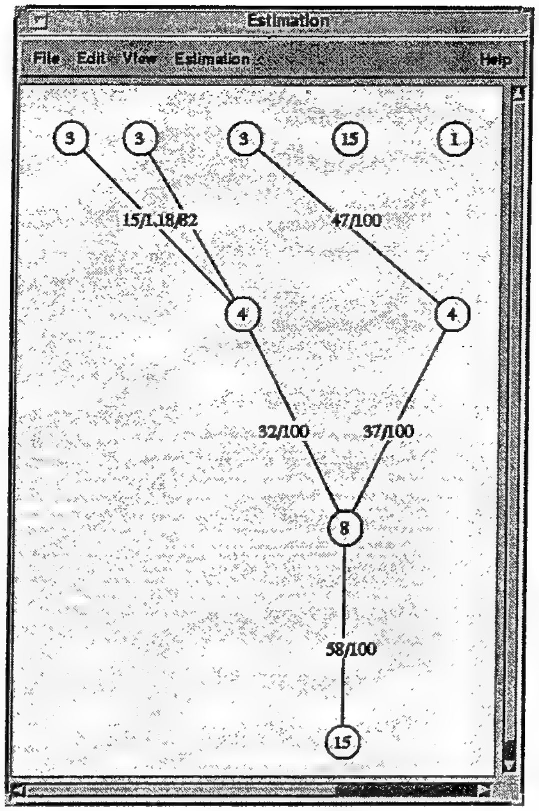


Figure 7.6: The object structure before clustering.

```
"ATM-McClintok"
```

7.1.4 Public Key Algorithm

We use one of public key ciphers [39] to ensure the security of data exchanged between ATMs and the Central Computer. Public key ciphers are asymmetric — not only the key used for encryption differs from that used for decryption, but it is also computationally infeasible to compute one key from the other. The term computationally infeasible implies that the computation would take a long time, even using the most powerful computers available.

The significant point about asymmetric ciphers is that the sender and receiver do not possess the same secret information. This asymmetric relationship can remove the requirement for the secure channel to exchange keys. The owner of the secret decryption key can freely distribute the associated nonsecret, or public, encryption key.

Any person holding the public encryption key can encrypt messages and then forward them securely to the holder of the secret decryption key. The holders of the public key cannot decrypt messages transmitted by other users of the public key, or indeed even their own encrypted messages.

Currently there are four public key ciphers in use:

- Rivest Shamir Adleman(RSA)
- Knapsack
- Discrete Log
- Elliptic Curve

The RSA cipher [40] has become the de facto standard for public key cryptography. Here, we use RSA cipher to encrypt and decrypt our data. The security of this algorithm lies in the difficulty of factoring large primes.

The steps in key generation, encryption and decryption are summarized for the sake of completeness [39].

- Key Generation:
  - Select any two large (e.g. 100-digit), unequal prime numbers, p and q;
  - Compute $r = p \times q$;
  - Compute $\phi(r) = (p-1)(q-1)$;
  - Select any integer d such that d lies between the maximum $\phi(p,q)$ and $r-1$ exclusively; moreover ensure that d and $\phi(r)$ have no common factors;
  - Find $e \equiv d^{-1}(\textbf{ modulo } \phi(r))$
  - Publish the public encryption key ($k_p = e$) and n, a parameter used in the encryption algorithm, while strictly maintaining the secrecy of the decryption key $k_s(= d)$ and the numbers p,q which are used in computing of d and e.
- Encryption:
  - Represent the original message in binary arithmetic and divide it into blocks such that the bit-string of the message blocks P_i's can be viewed as a 200-digit numbers.
  - The encrypted block C_i of P_i can be computed from the plaintext number P_i, by:

$$C_i \equiv P_i^e(\textbf{ modulo } r).$$

- Decryption:
  - The original message P_i is recovered by computing:

$$C^d \equiv P_i(\textbf{ modulo } r).$$

For example, Alice wants to send some data to Bob by using the public key system. Alice selects $p = 53$ and $q = 61$ giving $r = pq = 3233$ and $\phi(r) = (p-1)(q-1) = 3120$. Alices selects $d = 791$, hence $e = 71$, $(de = 791 * 71 = 56161 = 18 * 3120 + 1 =$

1 **modulo** 3120). Alice publishes modulus $r = 3233$ and public key (k_p), $e = 71$, maintaining the secrecy of $(k_s)d = 791$. Bob selects the message 1800 and encrypts it:

$$1800^{71} \equiv 2691 \textbf{ modulo } 3233.$$

Bob transmits the ciphertext block 2691 to Alice who decrypts it using her private decryption key 791:

$$2691^{791} \equiv 1800 \textbf{ modulo } 3233.$$

The key generation stage is computationally demanding. In addition, the steps in the key generation algorithm depend on the modular arithmetic operations involved in computing inverses and large exponents. Since the key generating operation is encapsulated inside the object "central computer", the operation is a good candidate for us to analyze fine grain parallelism.

For every transaction session, Central Computer computes a public key and both Central Computer and the associated ATM use this key to encrypt and decrypt the data. When next transaction comes, Central Computer will produce another public key for that session. Under this circumstance, we do not need to compute two huge primes with 200 digits long, and instead only need to compute two primes between 10000 and 65535 exclusively for this example.

7.1.5 Dynamic Behavior Analysis

The event trace diagram for the ATM system example is given in Figure 7.7.

With the event trace diagram, state transition description and the method description in the OOA result of the ATM example, we apply the dynamic behavior analyzing method decribed in Chapter 5 and find the following four pairs of objects which can be invoked concurrently:

- Customer, Central_Computer;
- ATM, Customer;
- Bank_America_Computer, Bank_One_Computer;
- Acount1, Acount2.

These analysis results can then be used in our object clustering algorithm.

7.2 The Air Force Defense System Example

We also use the hypothetical air force base defense system given in [8] to demonstrate our approach involving both coarse grain and fine grain parallelism. In this example,

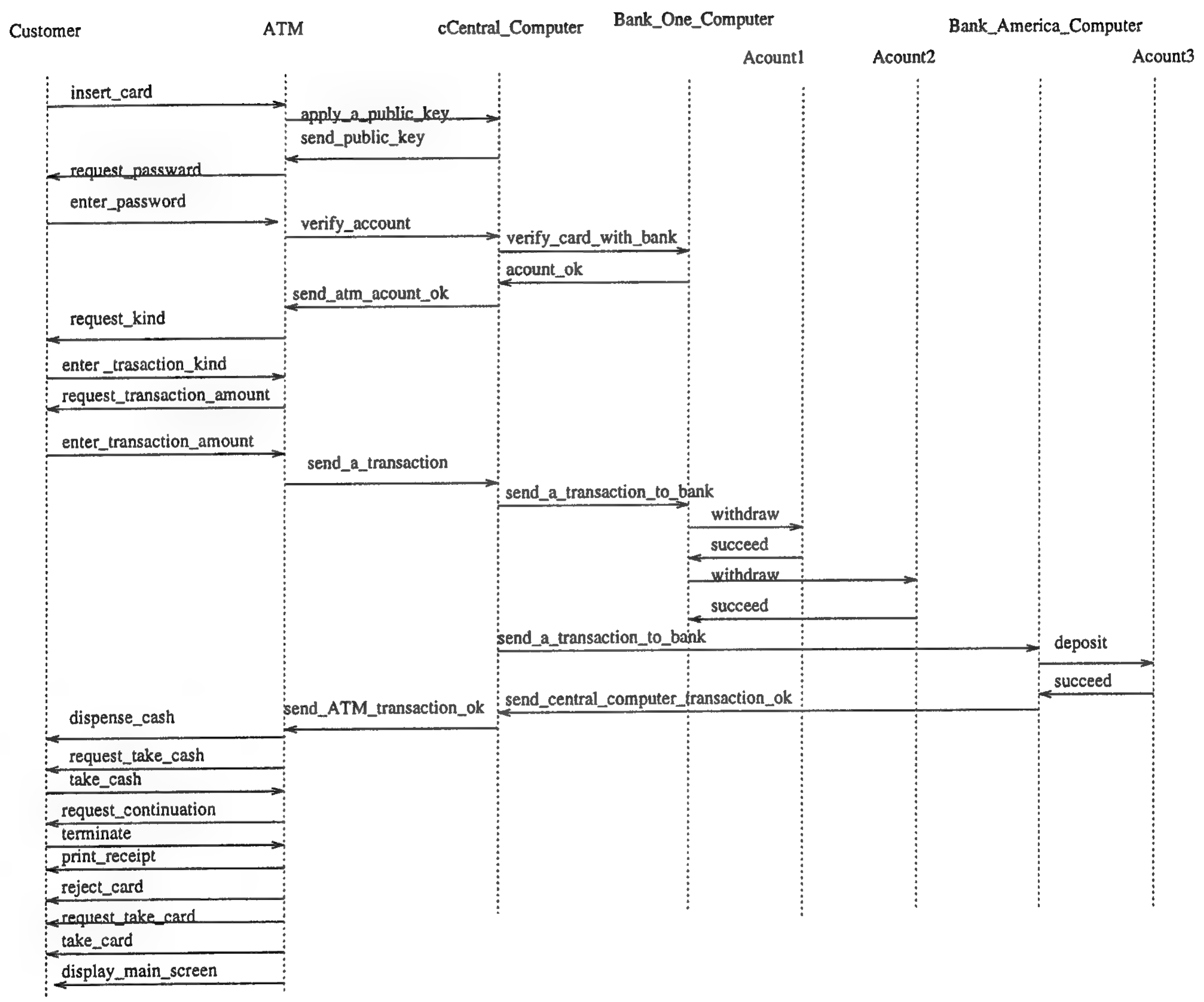


Figure 7.7: The event trace diagram for the transfer scenario of the ATM system example.

communication and synchronization aspects among air force bases were emphasized (coarse-grain parallelism). Here, we will focus on one of the bases and emphasize both communication and computation aspects.

7.2.1 Specifications of a Hypothetical Air Force Base Defense System

Assume that there are three air force bases that are closely connected. For the sake of simplicity, we assume that only one type of fighters, one type of bombers, one type of surface-to-air missile batteries for defensive purposes against the attacking enemy. Radars and C3 (Command, Control and Communication) facilities are available. Each base may have many radars, but the base gets only one correlated radar value. Each base will have several missile batteries and sufficient missiles to be used for its defense. Each base has either fighters or missiles for the defense. There would be one central C3I unit which advises each base as to what it should do for its defense. In our application, we will associate the C3I advice for a base along with the design of the base itself since this is a parallel processing system. This way, the commander at the center can know what is going on at different bases simultaneously and will also be able to give orders to different bases simultaneously.

For the example considered here, we will characterize one of the bases and emphasize more computation aspect on that base. The detailed description is as follows:

An air force base consists of radar installations, equipment, and armed personnel. The radar detects approaching hostile attacks on the base. It is assumed that the enemy cluster consists of either bombers or missiles, but not a combination of the two. The base, in turn, can use its fighters or its missiles, but not a combination of the two simultaneously for its defense. The defense strategy used by the base depends on the configuration of the enemy cluster.

Upon detection of an enemy cluster, the radar tracks the cluster to determine its composition. This enemy information, which is the number of bombers or missiles of each enemy cluster, is stored in a queue. The air force base retrieves the enemy information from the queue.

If the enemy cluster consists of x bombers, the base defends itself by launching either its fighters [represented by the computation of the function $F(x)$] or its missiles [represented by the computation of the function $G(x)$]. On the other hand, if the enemy cluster consists of y missiles, the base defends itself by launching its own missiles to intercept the incoming missiles [represented by the computation of the function $H(y)$].

In our implementation, we use a random number generator to simulate various incoming threats. For simplicity purpose, we assume that

$$F(x) = \sum_{i=1}^{x} i, \ i \epsilon I \tag{7.1}$$

$$G(x) = \pi \simeq \frac{1}{x}\sum_{i=1}^{x}\frac{4.0}{1.0+{\chi_i}^2},\ \chi_i = \frac{(i-0.5)}{x},\ i\epsilon I, \tag{7.2}$$

$$H(y) = \{z_1, z_2, z_3, ...\},\ z_i\epsilon\{Prime\ numbers\},\ 1 \leq z_i \leq y \tag{7.3}$$

Each of the defense strategies [the computation of $F(x)$, $G(x)$ and $H(y)$], and the radar will be executed in parallel on independent nodes to exploit *coarse grain parallelism.* Threats are added to a FIFO queue. The air force base removes a threat from the FIFO queue and computes either $F(x)$, $G(x)$ or $H(y)$ depending on the type of the threat.

Each of the defense strategies is executed on multiple processors to exploit *fine grain parallelism.* It does so by breaking down its task into smaller tasks which can then be executed in parallel on independent nodes. The results of these smaller tasks are then gathered together to yield the final result.

7.2.2 Object-Oriented Analysis

1. Identifying Classes and Objects

   We identify the following *Classes* from the requirement specification of the example:

   - Base – for air force base
   - Radar – for radar associated with Base.
   - Queue – for FIFO queue.

   From the requirement specification, we identify the following objects:

   - B – corresponding to a single air force base.
   - R – corresponding to the radar associated with the single air force base.
   - Q – for recording enemy cluster information.

2. Defining Class Interfaces

   Class interface of an object consists of the input and output parameters and their types. The class interfaces of the various objects identified in Section 7.2.1 are given at the end of this paragraph. As an example, let us consider the class *Queue*. We show one interface which is called method *Insert*. This method is invoked by the body of object *R*. From the domain knowledge of the example, we can infer that the radar value consists of the number of bombers or missiles attacking the base. The type of data is obviously integer. In a similar fashion, the class interfaces for the various classes can be determined. In order to illustrate the usage of global methods, the class interfaces for the classes *Base* and *Radar* consists of global methods, not class specific methods. The reason for using the global methods is that the computation, such as random number generator, finding prime numbers, π approximation, factorial summation, are all general operations which do not belong to any specific class. All class interfaces of this example are given below:

```
class Queue

   method Q_init(s:int -> Queue)

   method Insert ( -> Queue)

   method Assign (New:list -> Queue)

   method Delete ( -> Queue)

   method GetElem( -> int)

end class

class Base

   method IsPrime(number:int, factor:int -> int)
   method Prime(number:int -> int)
   method FindPrimes(low:int, upp:int, number:int -> int)

   method IntegerSum(l:list -> int)
   method FactSum(low:int, upp:int -> int)

   method RealSum(l:list -> real)
   method Pi(l:int, h:int, interval:int -> real)

end class


class Radar

   method Random(low:int, upp:int, number:int -> int)

end class
```

where Random, IsPrime, Prime, FindPrimes, IntegerSum, FactorialSum, RealSum, and Pi are global methods and do not belong to any specific class.

3. Specifying Dependency and Communication Relationships Among Objects

   Once the class interfaces are obtained for all the classes, we can establish the dependency and communication relationships among the objects from the object-oriented analysis phase. Figure 7.8 gives the dependency and communication

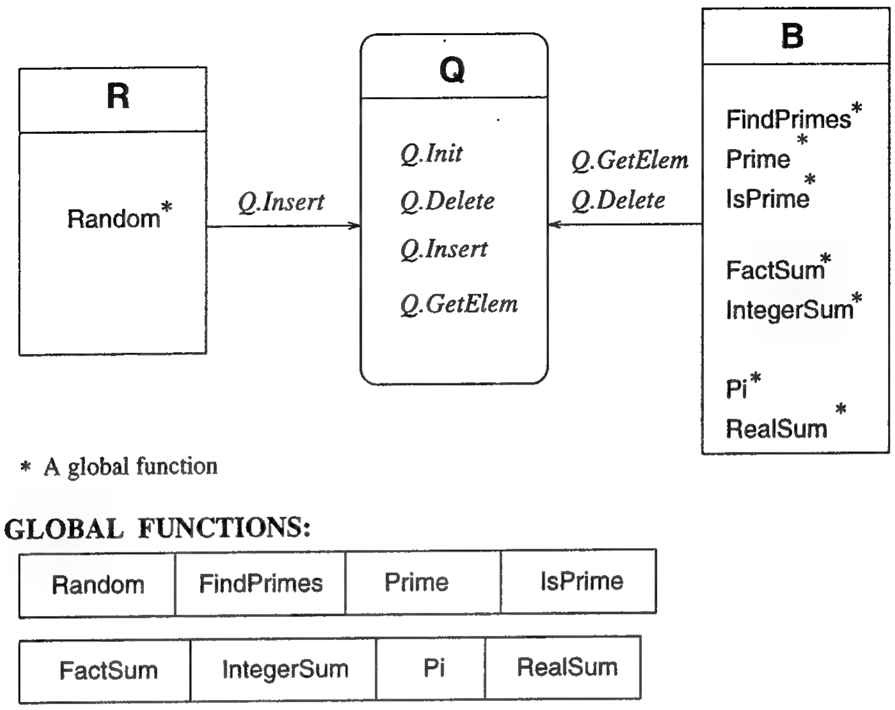


Figure 7.8: The object communication diagram for the set of decomposed objects of the hypothetical air force base defense example.

Table 7.2: Object classification of the hypothetical air force base defense example.

Classification	Objects
Active	R, B
Passive	Q
Pseudo-active	None

relationships among these objects. To illustrate the operation, let us consider the object R, which puts a radar value into the object Q. Thus, there exists communication between R and Q.

4. Identifying Active, Passive and Pseudo-Active Objects

   From the requirement specification and from the object communication diagram shown in Figure 7.8, we notice that the object R is not invoked by other objects, but does invoke other objects such as Q. Thus, R is identified as an active object. To illustrate the methods for identifying passive objects, let us consider the communication behavior of the object Q, which is invoked by other objects such as R and B, but never invokes any other objects. Such objects are classified as passive objects. If the communication behavior shows an object being invoked by other objects as well as invoking other objects, it is identified as pseudo-active object. Figure 7.8 shows no such object. Thus, we have no pseudo-active objects in this example. We can classify the objects as shown in Table 7.2.

5. Identifying Shared Objects

From the object communication diagram as well as the object behavior, we identify the object Q as a shared writable object. Shared writable objects are usually passive objects.

6. Specifying the Behavior of Each Object

   We are now in a position to describe the behavior of each object. For instance, let us consider the object R. The object R adds threats to a FIFO queue endlessly. Thus, we have the behavior of the object R. The behavior of each object is given below:

   Behavior of object R:

```
while(TRUE,
      (Q.Insert))
```

   Behavior of object B:

```
while(TRUE,
      let low = 1 in
      let upp = 10001 in
      let third = (/ (- upp low) 3) in
      let target = (Q.GetElem) in
         Q.Delete,
         if ( (null? target),
         # THEN do nothing because no threat to base
         # ELSE respond to threats
              if ( (<= target third),
                   # THEN-clause
                   (FindPrimes, 1, target)
                   # ELSE-clause
                   if ( (and (> target third) (<= target (* 2 third)) ),
                        # THEN-clause
                        (FactorialSum, 0, target),
                        # ELSE-clause
                        (Pi, 1, target, (- target 1))
      ))))
```

7. Identifying Bottleneck Objects

   We have identified the object Q as a shared writable object. Since different methods of this object are used by the objects R and B to access this object, the access to the object Q does not have to be serialized. Hence, Q is not a bottleneck object. Thus, we do not have any bottleneck objects in this example.

8. Checking for Completeness and Consistency of the Object-Oriented Analysis

   Because of the similariity of the example, by tracing through the behavior of the objects and looking at the class interfaces, we can see that the object-oriented analysis is complete and consistent.

7.2.3 Object Design

1. Establishing Hierarchy

   Since in this example we do not have two different types of objects with common behavior, we do not need to define a superclass. In other words, we do not have any inheritance in this example.

2. Designing Class Composition and Methods

   The class composition typically consists of local data present in the class. The type of data present in the class is also identified. In this stage, we also provide the methods present in each of the classes. As an example, consider the class composition of the class *Queue*. The data in the object is a list of integers generated by the global method *Random* and two integers. These constitute the class composition. In addition to these, we define the methods. The methods required for the class *Queue* are as follows:

   (a) Initialize the integers in the composition.
   (b) Add integers to the list *TargetsQ* in the composition.
   (c) Modify the list *TargetsQ* in the composition.
   (d) Delete the integers from the list *TargetsQ* in the composition.

   These are defined as follow:

```
Class Queue

   composition
      TargetsQ :  list
      seed     :  int
      number   :  int
   end composition


   method Q_init(s:int -> Queue)
      expression
         object Queue (seed = s, number = 0)


   method Insert ( -> Queue)
      expression
         let low = 1 in
         let upp = 10001 in
         let item = (Random low upp seed) in
         object Queue ( TargetsQ = (append_right TargetsQ item),
                        seed = item )
      method Assign (New:list -> Queue)
         expression
            object Queue ( TargetsQ = New )


      method Delete ( -> Queue)
         #guard (> number 0)
         expression
            object Queue ( TargetsQ = (tail TargetsQ),
                                    seed = seed )


      method GetElem( -> int)
         #guard (> number 0)
         expression
            (head TargetsQ)

   end class


   global

      method Random (low:int, upp:int, number:int -> int)
         expression
            let factor = (- (/ (+ low upp) 2) 13) in
            let x = (mod (* factor number) upp) in
            if ( (and (and (>= x low) (<= x upp)) (> x 0)),
                 (+ x 0),
                 (Random low upp x))


      method IsPrime(number:int, factor:int -> int)
         expression
             if ( (<= (* factor factor) number),
                  if ( (= (mod number factor) 0),
                       0,
                       (IsPrime number (inc factor)) ),
                  1 )


      method Prime(number:int -> int)
         expression
            if ( (or (= number 2) (= number 3)),
                 1,
                 if ( (= (IsPrime number 2) 1),
                      1,
                    0 ))


method FindPrimes(low:int, number:int -> list)
    expression
        (head while (lambda (x) (> (head (tail x))
                                   (head (tail (tail x)))),
                    lambda (x) (
                        let y = (head (tail x)) in
                        let z = (head (tail (tail x))) in
                        if ( (= (Prime y) 1),
                               [ (append_right (head x) y)
                                               (- y 1) z ],
                               [ (head x) (- y 1) z ] ))
                    ) [ [] number low ] )


method IntegerSum(l:list -> int)
    expression
        if ( (null? l),
               0,
               (+ (head l) (IntegerSum (tail l))) )


method FactSum(low:int, upp : int -> int)
    expression
        (head while (lambda(x) (< (head (tail x))
                                  (head (tail (tail x)))),
                    lambda(x)
                        let y = (head (tail (tail x))) in
                        [ (+ (head x) y) (head (tail x))
                                         (- y 1) ]
                        ) [ 0 low upp ] )


method RealSum(l:list -> real)
    expression
        if ( (null? l),
               0,
               (+ (head l) (RealSum (tail l))) )


method Pi(l : int, h : int, interval : int -> real)
    expression
        (head while(lambda(x) (< (head (tail x))
                                 (head (tail (tail x)))),
                        lambda(x)
                           let w = (/ 1.0 (head (tail
                                          (tail (tail
                                                 x))))) in
                           let t = (* (- (head (tail x)) 0.5) w) in
                           let tmp = (/ 4.0 (+ 1.0 (* t t))) in
                           [ (+ tmp (head x)) (+ 1 (head (tail x)))
                             (head (tail (tail x)))
                             (head (tail (tail (tail x)))) ]
                           ) [ 0 1 h interval ] )

end global
```

3. Designing the Body of the Objects

   The body of an object describes the control thread within the body. A control thread exists for only active and pseudo-active objects. Thus, the bodies exist for only active and pseudo-active objects. In our example, the bodies exist for the objects R and B since these objects have been identified previously as active objects. The behavior of the active objects should describe the body of that object. For example, the object R has a body which iteratively executes in accordance with its behavior specified before. In the following, we give the body of the active object.

   Body of object R:

```
while(TRUE,
      ;(
        R[| Q |] (Q.Insert),
        (delay 1)
        ))
```

   Body of object B:

```
while(TRUE,
      ;(
        let low = 1 in
        let upp = 10001 in
        let third = (/ (- upp low) 3) in
        let target = (Q.GetElem) in
        ;(
           R[| Q |] (Q.Delete),
           if ( (null? target),
                # do nothing because no threat to base
                (delay 2),
                ;(
                  # respond to threats
                  if ( (<= target third),
                ;( R[| B |] object Base (result =
                    (delta (FindPrimes, 1, target)))),
                   if ( (and (> target third)
                                (<= target (* 2 third)) ),
                         ;( R[| B |] object Base
                             (result = [(IntegerSum
                             (delta (FactSum, 0, target)))])),
                         ;( R[| B |] object Base
                             (result = (/ (RealSum
              (delta (Pi, 1, target, (- target 1) )))
                                       (- target 1) )))
              ))
         ))
)))
```

Chapter 8

Discussion

In this project, we have completed the following tasks:

- Investigation on object identification methods
- Development of object-oriented analysis tool
- Development of communication estimation and object clustering tool
- Investigation on parallelism analysis
- Design and implementation of the back-end translator on cluster of workstation.

While existing approaches focus on developing software for scientific computation, our approach is suitable for general large-scale software development for parallel processing systems. Our approach is architecture-independent, and thus the programmers are free from explicitly specifying synchronization and communication. The graphical user interfaces of our tools can help software developer capture the overall pictures of the class/object hierarchy, design and modify class/object interactively.

Our approach exploits coarse grain parallelism by deriving all the concurrent objects from a problem and classifying them into different categories: active, pseudo active or passive. Our approach can easily be applied to general communication-oriented problems in which a number of objects need to be executed simultaneously, and these objects interact with one another periodically. It has been applied to software development for distributed computing systems [6]. The ATM system and air force base defense example in Chapter 7 are two communication-oriented and computation-oriented applications. The synchronization among different objects, such as the radars and the bases, has been realized by using guard structures within object methods. Method invocations fulfill communication among different objects.

Our approach exploits fine grain parallelisms at the method level. Parallel functions specify data or functional parallelisms in a method. It is suitable for computation-oriented applications, such as the ATM system example, where we embed all the computations inside the Central_Computer object's methods and distribute these computations on different nodes. Table 8.1 shows the average execution time of each transaction from the 100-transaction loop in the ATM system example. The corresponding speed-up curves are shown in Figure 8.1. It is noted that these curves approach very closely to the ideal speed-up line when the width of public key is increasing. Table 8.2 shows the execution time for the air force base defense example in 100 iterations (the program itself goes infinitely) using nCube C directly. Table 8.3 shows the same application using PROOF/L run on the nCube through the PROOF/L back-end translator. The corresponding PROOF/L code running on our 8-workstation cluster is shown in Table 8.4. The time scale is in seconds. (All the examples use the same PROOF/L source code. The execution time data on nCube are from last project.)

In order to make our approach more practical, we need to improve our approach in the following aspects:

- Conforming to standards: Standardize the object-oriented analysis and design results.
  Our user interface specification language can capture most of object-oriented features in the specification, but it lacks the ability to interoperate with other languages. There are two ways to overcome this problem. We can transform the object-oriented result to some well recognized representation such as Unified Modeling Language (UML, follow the link "http://www.rational.com/uml" for details).
- Fully implementation of the CASE tools: Since we have only built prototype tools for the project, the CASE tools should be fully developed and tested for practical use.
  - Semantic analysis tool. This tool currently does not support diagram drawing and graphical representation output. Future version should include this feature and be more user-friendly. Furthermore, we should base our concurrecy identification, design implementation on UML, which will facilitate our approach and tool to be adopted for other environments.
  - Concurrency identification tool. We will extend this tool with a GUI to let the user graphically view the analysis result. We can also extend our basic model to include both sychronous and asynchronous object communication.
  - Communication estimation and object clustering tool. Currently, the only output device for this tool is the screen. We can extend our output functions to produce graphical files, e.g., postscript file, and print the object hierarchy and clustering result directly on printers or plotters.
- Extension to distributed computing systems: Currently, our approach is designed for software development for parallel processing systems, but it can be extended for distributed computing systems. To reduce the development effort and increase the interoperability, we should adopt some standards, like CORBA or Active X, and refine the load balancing algorithm.

Table 8.1: The average time of one transaction from 100 transactions loop (in seconds) in the ATM system example programmed in PROOF/L and translated to PVM/Sun C using different numbers of workstations.

	Number of Workstations							
Digits of Key	1	2	3	4	5	6	7	8
4	2.86	1.55	1.23	1.19	1.25	1.27	1.27	1.28
5	23.00	12.01	8.10	6.20	6.00	5.00	4.00	4.00
6	215.00	108.32	75.50	56.09	45.45	38.00	33.66	29.21

Table 8.2: The execution time of the hypothetical air force base defense example programmed directly in nCube C using different numbers of nodes.

Num of Nodes Used	Air Force base defense (sec)
4	639.89
8	417.47
16	254.26
32	150.67
64	115.05

Table 8.3: The execution time of the hypothetical air force base defense example programmed in PROOF/L and then translated to nCube C using different numbers of nodes.

Num of Nodes Used	Air Force base defense (sec)
4	1135.68
8	592.25
16	385.73
32	314.15
64	240.98

Table 8.4: The execution time of 100 iteration of the hypothetical air force base defense example programmed in PROOF/L and then translated to PVM/Sun C using different numbers of nodes (Sun workstations).

Num of Nodes Used	Air Force base defense (sec)
1	218.68
2	109.25
4	57.73
6	39.15
8	29.98

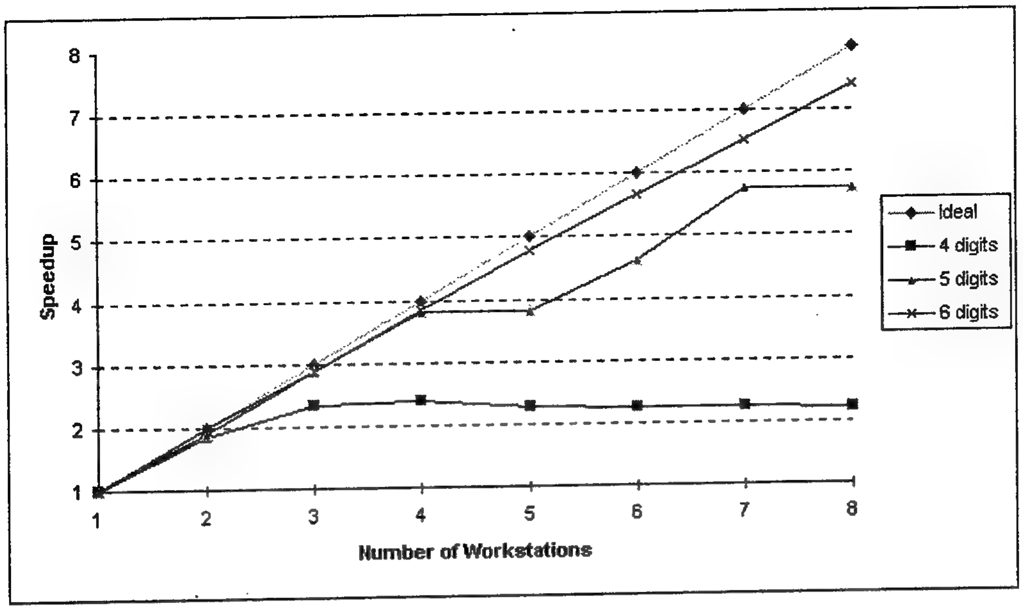


Figure 8.1: Speedup using various numbers of workstations to simulate ATM system in PROOF/L.

Appendix A

The User Interface Description Language Used in our Object-Oriented Analysis Tool

The following are the formal syntax and semantics for the user interface description language used in our object-oriented analysis tool.

A.1 The Syntactic Definition

```
UIDL -> TEXT | "<UIDL>"  ENTITYLIST EXTRA "</UIDL>"

EXTRA -> "<extra>" ENTITYLIST "</extra>"

ENTITYLIST -> ENTITY ENTITYLIST | ENTITY

ENTITY -> CLASS | OBJECT | ATTRIBUTE | METHOD | ASSOCIATION |
  AGGREGATION | INHERITANCE | EVENT | TEXT | COMMENT

CLASS -> "<class>" NAME ID CROLE PERSISTENCE DESC "</class>"

OBJECT -> "<object>" NAME ID OROLE PERSISTENCE PARAMS DESC "</object>"

ATTRIBUTE -> "<attr> NAME ID TYPE DESC "</attr>"

METHOD -> "<method>" NAME ID PARAMS ID DESC </method>

ASSOCIATION -> "<assoc>" NAME ID ID "(" CARD ")" ID DESC "</assoc>"

AGGREGATION -> "<aggre>" NAME ID ID DESC "</aggre>"
```

```
EVENT -> "<event>" NAME ID ID COUNT DESC "</event>"

STATE -> "<state>" NAME ID ID DESC "</state>"

INHERITANCE -> "<inhe>" IDLIST ":" ID "</inhe>"

PARAMS -> "(" PARAMLIST ")"  | "(" ")"  |

PARAMLIST -> PARAMLIST ";" PARAM | PARAM

PARAM -> TEXT ":" TYPE

CARD -> ID ":" ID

IDLIST -> ID | IDLIST ID

CROLE -> "actor" | "agent" | "server"

PERSISTENCE -> "transient" | "persistent"

OROLE -> "active" | "pseudo-active" | "passive"

TYPE -> "int" | "real" | "string" | "list" | "array" | ID |

COMMENT -> "<!" TEXT ">"

NAME -> QSTRING

DESC -> QSTRING

QSTRING -> "\"" TEXT "\""

TEXT -> ALPHADIGIT*

COUNT -> DIGIT*

ID -> DIGIT*

ALPHADIGIT -> all the ascii symbols except "<" ">" "!" "\" ":"
              | "\<" | "\>" | "\!" | "\\" | "\:"

DIGIT -> 0|1|2|3|4|5|6|7|8|9

PUNC -> [.,!?]
```

```
WS -> [\t ]*

NL -> [\n]*
```

A.2 The Semantic Description

Each entity in terms of class, object, attribute, method, association, aggregation, inheritance, event, and state has the following semantic meaning:

```
a). Classs:
      <class> name : id : role (actor, agent, server) : persistence
              (transient, persistent) : description </class>

b). Class instantiation:
      <object> name : id : class id : role (active, pseudo-active,
               passive) : ( instantiated parameters ) : description </object>

c). Attribute:
       <attr> name : id : class id : type : description </attr>

d). Method:
      <method > name : id : class id : (parameters <arg : type>,*) :
                description </method>

e). Association:
      <assoc> name : id : source class id : destination class id :
              cardinality : persistence (if cardinality is n:m) :
              description </assoc>

f). Aggregation:
      <aggre> name : id : type class id : container class id :
              description </aggre>
        Aggregations can be viewed as special attributes of the container
        class or an individual class.

g). Inheritance:
      <inhe> (subclass id ,)* subclass id : superclass id : name : id :
             description </inhe>
         It only can be presented after subclass and superclass have
         been defined.

h). Event:
      <event> name : id : source class id : destination class id :
              sequence : description </event>
```

i). State:
    <state> name : id : class id : description < /state>

j). Plain text: text

k). <extra> entity list </extra> to denote any object-oriented terms not included in the problem statements.

Bibliography

[1] J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM.* vol. 21, No. 8, Aug. 1978, pp. 613–641.

[2] R., Milner, "A Proposal for Standard ML," *Proc. of 1984 ACM Conference on LISP and Functional Programming.* 1984, pp. 184–197.

[3] J. McGraw, *et al, SISAL: Streams and Iteration in a Single Assignment Language,* Language Reference Manual Version 1.2, 1985.

[4] S. S. Yau, X. Jia and D.-H. Bae, "PROOF: Parallel Object-Oriented Functional Computation Model," *Journal of Parallel and Distributed Computing,* Vol. 12, No. 3, July, 1991, pp. 202-212.

[5] S. S. Yau, X. Jia, D-H. Bae, M. Chidambaram, and G. Oh, "An Object-Oriented Approach to Software Development for Parallel Processing Systems," *Proc. 15th Int'l Computer Software & Applications Conf. (COMPSAC 91),* September 1991, pp. 453-458.

[6] S. S. Yau, D.-H. Bae and M. Chidambaram, "A Framework for Software Development for Distributed Parallel Computing Systems, " *Proc. Third Workshop on Future Trends of Distributed Computing Systems,* April 1992, pp. 240–246.

* [7] S. S. Yau, D.-H. Bae, M. Chidambaram, G. Pour, V. R. Satish, W-K. Sung and K. Yeom, *Software Engineering For Effective Utilization of Parallel Processing Computing Systems,* Final Technical Report RL-TR-93-113, Rome Laboratory, Air Force Material Command, Griffiss Air Force base, New York, June 1993.

[8] S. S. Yau, D.-H. Bae, P. K. Gupta, S.-I. Paek, T. J. Thigpen, J. Wang and M. A. Wells, *A Software Development Methodology for Parallel Processing Systems,* Final Technical Report RL-TR-95-190, Rome Laboratory, Air Force Material Command, Griffiss Air Force base, New York, October, 1995.

[9] G. Booch, *Object-Oriented Analysis and Design,* Benjamin/Cummings, 1994.

[10] S. Shlaer, and S. J. Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data,* Prentice Hall, 1988.

[11] P. Coad and E. Yourdon, *Object-Oriented Analysis,* Yourdon Press, 1991.

*RL-TR-93-113 is Distribution Limited to U.S. Government Agencies Only.

[12] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[13] S. S. Yau, D.-H. Bae and J. Wang, "An Architecture-Independent Software Development Approach fro Parallel Processing System", *Proc. of the 19th Annual Int'l Computer Software & Applications Conf. (COMPSAC 95)*, August, 1995, pp. 370-375.

[14] S. S. Yau and J. Wang, "A Framework for an Integrated Tool Set for Object-Oriented Software Development", *Proc. of the 20th Annual Int'l Computer Software & Applications Conf. (COMPSAC 96)*, August, 1996, pp. 502-507.

[15] K. S. Rubin and A. Goldberg, "Object Behavior Analysis," *Communications of the ACM,* September 1992, Vol. 35, No. 9, pp. 48–62.

[16] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[17] D. E. Eager J. Zahorjan and E. D. Lazowska, "Speedup Versus Efficiency in Parallel Systems," *IEEE Trans. on Computers*, Vol. 38, No. 3, 1989, pp. 408–423.

[18] H. S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Trans. on Software Engineering*, Vol. SE-3, No. 1, 1977, pp. 85–93.

[19] C. C. Shen and W. T. Tsai, " A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion," *IEEE Trans. on Computers*, Vol. 34, No. 3, 1985, pp. 197–203.

[20] W. W. Chu, L. J. Holloway, M.-T. Lan, and K. Efe, "Task Allocation in Distributed Data Processing," *IEEE Computer*, Vol. 13, No. 11, 1980, pp. 57–69.

[21] O. I. El-Dessouki and W. H. Huan, "Distributed Enumeration on Network Computers," *IEEE Trans. on Computers*, Vol. C-29, No. 9, 1980, pp. 818–825.

[22] K. Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *IEEE Computer*, Vol. 15, No. 6, 1982, pp. 50–562.

[23] S. S. Yau and I. Wiharja, "An Approach to Module Distribution for the Design of Embedded Distributed Software Systems," *Information Sciences*, Vol. 56, 1991, pp. 1–22.

[24] S. S. Yau, D.-H. Bae, and Gilda Pour, "A Partitioning Approach for Object-Oriented Software Development for Parallel Processing Systems," *Proc. 16th Annual Int'l Computer Software & Applications Conf. (COMPSAC 92)*, October 1992, pp. 251–256.

[25] S. S. Yau and V. R. Satish, "A Task Allocation Algorithm for Distributed Computing Systems," *Proc. 17th Annual Int'l Computer Software & Applications Conf. (COMPSAC 93)*, November 1992, pp. 336–342.

[26] C. N. Nikolaou and A. Ghafoor, "On the Assignment Problem of Arbitrary Process Systems to Heterogeneous Distributed Computer Systems," *IEEE Trans. on Computers*, Vol. 41, No. 3, March 1992, pp. 257–273.

[27] D. Fernandez-Baca, "Allocating Modules to Processors in a Distributed Systems," *IEEE Trans. on Software Engineering*, Vol. 15, No. 11, November 1989, pp. 1427–1436

[28] S. M. Shatz, J-P Wang and M. Goto, "Task Allocation for Maximizing Reliability of Distributed Computer Systems," *IEEE Trans. on Computers*, Vol. 41, No. 9, September 1992, pp. 1156-1168.

[29] S. S.Yau, D.-H.Bae and K. Yeom, "An Approach to Object-Oriented Requirements Verification in Software Development for Distributed Computing Systems", *Proc. 18th Int'l Computer Software & Applicattions Conf.(COMPSAC 94)*, November 1994, pp. 96-102.

[30] L. B. Protsko, P. G. Sorenson, J.-P. Tremblay, and D. A. Schaefer, "Towards the Automatic Generation of Software Diagrams," *IEEE Trans. on Software Engineering*, Vol. 17, No. 1, January 1991, pp. 10-21.

[31] C. Batini, E. Nardelli, and R. Tamassia, "A Layout Algorithm for Data Flow Diagrams," *IEEE Trans. on Software Engineering*, Vol. SE-12, No. 4, April 1986, pp. 538-546.

[32] E. R. Gansner, S. C. North, and K. P. Vo, "DAG – A Program that Draws Directed Graphs," *Software–Practice and Experience*, Vol. 18, No. 11, November 1988, pp. 1047-1062.

[33] S. S. Yau, K. Yeom, B. Gao, L. Li and D-H. Bae, "An Object-Oriented Software Development Framework for Autonomous Decentralized Systems," *Proc. Second Int'l Symposium on Autonomous Decentralized Systems (ISADS 95)*, 1995, pp. 405–411.

[34] G. A. Miller, "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *Psychological Reviews*, Vol. 63, March 1965, pp. 81-97.

[35] J. J. Dongarra and T. Dunigan, "Message-Passing Performance of Various Computers", Available electronically on the PVM home page, the URL is http: //www.netlib.org/utk/papers/commperf.ps, August 1995, page 14.

[36] E. Yourdon, "Object-Oriented System Design", Yourdon Press, 1994.

[37] nCUBE Inc., *nCUBE 2 Programmer's Manual*, 1992.

[38] A. Geist, Adam Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. SUnderam, *PVM: Parallel Virtual Machine A users' Guide and Tutorial for Networked Parallel Computing*, The MIT Press, 1994.

[39] W. Caelli, D. Longley and M. Shain, "Information Security Handbook", Stockton Press, 1991.

[40] R. Rivest, A. Shamir and L. Adelman, "A method for obtaining digital signatures and public-key cryptosystems", *Communications of the ACM,* Vol. 21, No. 2, Feb. 1978, pp. 120-128.

☆U.S. GOVERNMENT PRINTING OFFICE: 1998-610-130-61157